

TI-Basic Developer

The TI-Basic Information Repository

The Calculators

This section is concerned with the TI-83 family of graphing calculators, which all use the same base processor chip — a Zilog Z80. There are five different calculators that are within this group: TI-83, TI-83+, TI-83+SE, TI-84+, and TI-84+SE. Each of these calculators has their own features and unique qualities.

fold

Table of Contents

- [The TI-83 Calculator](#)
- [The TI-83+ Calculator](#)
- [The TI-83+SE Calculator](#)
- [The TI-84+ Calculator](#)

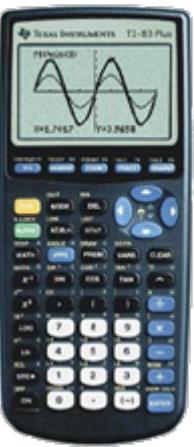
The TI-83 Calculator



The TI-83 is the oldest calculator in the group, being released back in 1996. It is designed to be an upgrade from the TI-82, featuring a sleeker case design, more memory (27K bytes of RAM), and a faster processor (6MHz). It kept some of the features the same as the TI-82, such as the screen size and being powered by 4 AAA batteries, to allow for backwards compatibility with the TI-82.

This means that while some of the TI-Basic commands on the TI-83 have a different syntax, at the core the TI-83 can execute the TI-82's TI-Basic programs. Some of the differences between TI-Basic for each calculator are how math is interpreted (implied multiplication versus regular multiplication) and commands that have an opening parentheses attached to the end of them (such as the trigonometry commands). When it comes to assembly programs, however, the TI-83 cannot execute the TI-82's assembly programs without some modification because of the processor upgrade.

The TI-83+ Calculator

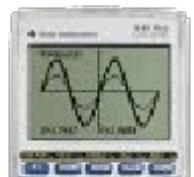


The TI-83+ was released in 1999, and it was meant as an upgrade from the TI-83 with more memory and a faster processor. At the same time, it kept several features of the TI-83 to maintain backwards compatibility: the case design, the screen size (16x8 home screen and 96x64 graph screen), and the link port. There are some major differences, however.

The TI-83+ cannot run assembly programs made for the TI-83 because it uses a different format: there are three built-in commands ([AsmPrgm](#), [AsmComp](#), and [Asm\(\)](#)) used for running assembly programs, while the TI-83 has no such commands. While the TI-83 uses a 6MHz processor, the TI-83 Plus uses a speedier 8MHz processor. It should be noted, however, that the TI-83+ only runs at the 6MHz speed unless altered by an assembly program.

The TI-83 comes with 24K bytes of available memory built-in, while the TI-83+ comes with 184K bytes of available memory: 24K bytes are RAM and 160K bytes are archive memory, or more commonly called "Flash" memory. Archive memory allows you to store data, programs, applications, or any other variables to a safe location where they cannot be edited or deleted inadvertently from a RAM crash. This creates a compatibility issue with TI-83 TI-Basic, however, because of the use of the [Archive](#) and [UnArchive](#) commands that are on the TI-83+.

The TI-83+SE Calculator



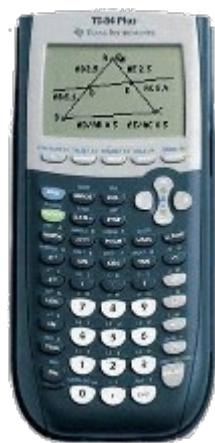
The TI-83+SE (short for Silver Edition) was the next calculator upgrade in the group. When it was released by TI in 2001, it became instantly popular because of its unique look and increased memory and speed. However, TI has since decided to stop production of it and focus on the TI-84+SE calculator instead. The TI-83+SE has now become somewhat of a collector's item.



After seeing the success of the TI-83 calculator series, TI decided to give their next TI-83 series calculator a unique look to set it apart from the other TI-83 calculators. The TI-83+SE calculator look consists of a transparent silver case with silver sparkles sprinkled throughout. What really made the calculator shine, though, was that the transparent case allowed you to see what the internals of the calculator looked like without even having to open up the calculator.

In addition to the unique look, TI also decided to upgrade the memory and speed. While maintaining almost complete backward compatibility, the TI-83+SE features 128K bytes of RAM and 1.5M bytes of archive memory. It should be noted, however, that only 24K bytes of RAM are available to TI-Basic programmers (you need to use assembly to access all 128K bytes). The TI-83+SE uses a 15MHz processor, but it can also be made to run at the 8MHz and 6MHz speeds through assembly.

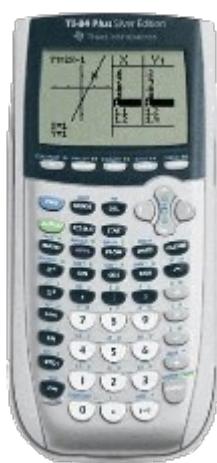
The TI-84+ Calculator



The TI-84+ was released in 2004, and it was meant to be an upgrade of the TI-83+. The TI-84+ introduces a couple new things to the TI-83 calculator series: a built-in clock and a mini USB link port. The built-in clock can be used in TI-Basic by using the new clock commands that go with it, while the mini USB link port greatly increases the speed of linking the calculator to a computer.

The TI-84+ also improves upon the TI-83+ in terms of memory and speed: 24K bytes of RAM and 480K bytes of archive memory; and a 15MHz processor (the same one that the TI-83+SE has). The most obvious change that the TI-84+ brings is a completely new case design. Gone are the slightly rounded edges and appearance; in its place is an almost circular look from the front, with the edges smoothly flowing around to the back of the calculator.

The TI-84+SE Calculator



The TI-84+SE was released along with the TI-84+ in 2004, kind of like a TI-83 calculator series package upgrade. The TI-84+SE was meant to be an upgrade of the TI-83+SE, and it includes the same upgrades that the TI-84+ got. The two new innovations that the TI-84+SE introduces are: interchangeable faceplates and a kickstand; these things are basically optional add-ons for your calculator.

The interchangeable faceplate can be useful if you want to change the front look of your calculator. You simply purchase a different faceplate and swap it with the current faceplate. The calculator's initial faceplate is a light gray/silver. The kickstand is built into the calculator lid, and it allows you to set the calculator to four different viewing angles. Concerning memory and speed, the TI-84+SE has the same amount of memory and speed as the TI-83+SE.

Calculator Comparison

Model	Processor	RAM	ROM	Screen Size	Link Port	Clock	Release Date
TI-83	6 MHz	27 KB	None	96x64	I/O	No	1996

TI-83+	6 MHz	24 KB	160 KB	96x64	I/O	No	1999
TI-83+SE	15 MHz	24KB (128 KB)	1.5 MB	96x64	I/O	No	2001
TI-84+	15 MHz	24 KB	480 KB	96x64	I/O+USB	Yes	2004
TI-84+SE	15 MHz	24KB (128 KB)	1.5 MB	96x64	I/O+USB	Yes	2004

Known ROM Versions

TI occasionally releases updates to the ROM version for each calculator, which either fix existing bugs, improve calculator performance, or add new commands and functionality. You can check the ROM version on your calculator by selecting the About option in the Memory menu, which is accessible by pressing 2nd MEM. See [portability](#) for a list of changes in functionality between the OS versions.

Model	Known ROM Versions
TI-83	1.02, 1.03, 1.04, 1.06, 1.07, 1.08, 1.10
TI-83+	1.03, 1.06, 1.08, 1.10, 1.12, 1.13, 1.14, 1.15, 1.16, 1.17, 1.18, 1.19
TI-83+SE	1.13, 1.14, 1.15, 1.16, 1.17, 1.18, 1.19
TI-84+	2.21, 2.22, 2.30, 2.40, 2.41, 2.43
TI-84+SE	2.21, 2.22, 2.30, 2.40, 2.41, 2.43

Pre-Loaded Applications

Except for the TI-83 which has no Flash ROM, all of the other TI-83 series of calculators come with some pre-loaded applications for users to use. Since most of the applications are rather limited in use and scope, not to mention that they each take up 16K bytes or more of memory, most people end up deleting them off of their calculator to allow them to fit more programs and games. If you want to put them back on your calculator again, you can find them all on [TI's website](#).

Model	Pre-Loaded Applications
TI-83+	Language Localization, Probability Simulation, Science Tools, StudyCards, Vernier EasyData
TI-83+SE	CellSheet, GeoMaster, Language Localization, Organizer, Periodic Table, StudyCards
TI-84+	Cabri Jr., Conic Graphing, Inequality Graphing, Language Localization, LearningCheck, LogIn, Probability Simulation, Science Tools, StudyCards, TI CBL/CBR, TI me Span, Topics in Algebra 1, Transformation Graphing, Vernier EasyData
TI-84+SE	Area Formulas, Cabri Jr., Catalog Help, CellSheet, Conic Graphing, Fundamental Topics in Science, GeoMaster, Inequality Graphing, Language Localization, LearningCheck, LogIn, NoteFolio, Organizer, Periodic Table, Polynomial Root Finder and Simultaneous Equation Solver, Probability Simulation, Puzzle Pack, Science Tools, Start-Up Customization, StudyCards, TI CBL/CBR, TI me Span, Topics in Algebra 1, Transformation Graphing, Vernier EasyData

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/thecalcs>

Why TI-Basic?

TI-Basic is the built-in programming language of the TI graphing calculators. You can create TI-Basic programs on the computer using the Graph Link or TI Connect software, or on the calculator itself through the program editor (see the [starter kit](#) for more information).

Knowing TI-Basic is important because it is one of the main ways that people use their calculators; if you are unable to program in TI-Basic, you will not be able to effectively communicate with others concerning your calculator.

Advantages of TI-Basic

There are several advantages of programming your calculator in TI-Basic. First, and foremost, it is the most well known calculator programming language.

With most high schools requiring TI graphing calculators for math and science classes, TI-Basic is often used by students to make small math or science programs. For many of these students, TI-Basic is the first programming language they have ever used.

Second, TI-Basic is extremely simple to learn. In TI-Basic, most of the commands are easily understood. The commands are written in plain English or easily comprehended abbreviations: Disp, Dec, etc. In addition, the commands are generally self-explanatory. For example, it is not very hard to recognize that the [Pause](#) command pauses a program.

Related to the simplicity of learning, the third advantage of TI-Basic is that it is the only language (so far) that can be programmed directly on the calculator. [Assembly](#) programs need to be written on a computer, and then converted into machine code with an assembler and several other programs. These programs are currently only available on computers.

The next advantage of TI-Basic is that it is very easy to do calculations in. Though TI-Basic can be used to write games as well, it's really useful for math programs. A math program in another language would probably have to call the same routines that TI-Basic uses anyway; this would be much more complicated, and wouldn't be an improvement in size or speed.

Lastly, if you mess up in TI-Basic (i.e., your program has an error), it just gives you an [error message](#). If an assembly program has an error, however, the results wouldn't be as good. Depending on the severity of the error, you can cause your calculator's RAM to be cleared, or even leave your calculator in an endless loop, rendering it completely useless. TI-Basic does not have that problem, because no matter where you are in a TI-Basic program, you just have to press the ON key to stop execution.

Disadvantages of TI-Basic

TI-Basic does have some disadvantages. Its main disadvantage is its speed. Because TI-Basic is converted by the calculator into machine code before it is executed, it loses much of its speed. Doing anything involving calculations or [graphics](#) is quite slow in TI-Basic. Really, the speed of TI-Basic comes nowhere close to the speed of assembly. You just need to play an assembly game (such as Super Mario) to see the great difference in speed.

TI-Basic History

Texas Instruments has included TI-Basic support with each graphing calculator (starting with the TI-81), and the TI-Basic language has evolved along with the calculators (adding new features and functionality).

With the release of the TI-84+/SE calculators, TI-Basic was enriched with [time and date](#) commands that use the new built-in clock, as well as some additional [statistics](#) commands.

The other disadvantage of TI-Basic is that it does not have low-level access to the calculator's hardware. While this is intentionally done to prevent potential misuse, it has the result of limiting the quality of TI-Basic programs. This is mainly a problem with input (the `getKey` command is limited to one key at a time) and graphics (the drawing functions are just simple pixels, lines, and circles).

The fact that the TI-Basic syntax is not very strict, is also a disadvantage. Using TI-Basic-like optimizations in other programming languages, is not a very good idea. Optimize your calculator programs, but don't fall into bad programming habits!

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/whytibasic>

Using This Guide

It is the goal of this site to eventually cover all information on TI-Basic programming for the TI-83 series of calculators. However, an excess of information can be overwhelming, so this page gives a suggested reading order (as well as necessary tidbits) so you don't get lost.

There are two alternatives for those who have just ventured into programming for these calculators. While you might want to just jump in, we have a tutorial that explains the basics of TI-Basic and another that answers basic TI-Basic questions.

The first, the [TI-Basic Starter Kit](#), teaches TI-Basic without assuming any previous programming experience. In fact, the very first section explains how to create your first program. After reading this tutorial carefully, you should be more than ready to handle the rest of this site.

The second tutorial is the [TI-Basic FAQ](#). As the name suggests, it is an attempt to answer the common TI-Basic questions that people ask. Many of the questions are related to each other, so it is recommended that you read through the whole list. If you have any questions that aren't mentioned on the list, or an answer doesn't help, please leave a post in the [forums](#) and somebody will try to help you.

Further Reading

At this point, you should be familiar with more than a few TI-Basic commands. It might be a good idea to just jump into the [command index](#) and click on commands that sound interesting — you can really improve your TI-Basic knowledge that way. Or, select a category from the Commands menu in the top navigation, and read about commands in a more general way.

Looking at code examples is also a good way to learn. [Games](#) and [Programs](#) are a good place to find such example code. See the [Routines](#) page for several short routines to get simple things done in the best possible way.

Use the [glossary](#) whenever you come across a term you're not familiar with. If it's not there or the entry doesn't help, drop a note in the [forums](#) and (usually within a day) helpful people will explain it and hopefully improve the glossary as well, so no one else has the same trouble.

The pages in the Reference section of the top menu are probably not ones you'd read for fun.

Getting Additional Help

Most of the members of this wiki can be found on the [forum](#), so you can post there with questions and get help with whatever you are working on or trying to learn.

In order to post on the forum, however, you need to [create](#) an account. This is an easy process, and should take no more than a couple minutes.

Take a look at them at any time to see what they're all about, and then check back when you need to know more.

Writing a Program

Of course, the best way to learn these topics is to come up with a project of yourself (check the [projects](#) page if you have no ideas), and go through the steps as you're doing it:

1. Look at [Planning](#) when you're thinking about how to approach the problem.
2. Consider [Commenting Code](#) and [Code Conventions](#) when you're writing it.
3. If bugs arise (and they most likely will), see the section on [Debugging](#).
4. When the program works, add [Setup](#) and [Cleanup](#) to it, and check the sections on [Usability](#) and [Portability](#).
5. If the program is too slow (and maybe even when it's not), see [Optimization](#) and [Code Timings](#) for ways to improve it.
6. Finally, see [Releasing Your Program](#) for how to earn TI-Basic programming fame by making the program public.

These are listed in the Design section of the side navigation menu.

Advanced Topics

The Techniques section in the side navigation menu discusses some advanced issues in TI-83 programming. You should probably have a good grasp of programming before venturing into these pages, but they are worth reading. Give them a glance to see how much you can understand.

Each technique is mostly a stand-alone page. Here are the relative difficulties of the pages:

Easy	Intermediate	Hard
<ul style="list-style-type: none">• Friendly Windows — makes using the graph screen commands much easier.• Piecewise Expressions — very important to programmers.• Saving Games — almost as easy as just storing to a variable.• Highscores — an extension of saving, using a string for names and a list for scores.• Animation — adds some visual pop or pizazz to your programs.	<ul style="list-style-type: none">• Validation — how to ensure user input satisfies your requirements.• Making Maps — how you store the contents of the screen to a variable.• Movement in Maps — adds user interaction to programs.• Custom Text Input — useful when you want to get input on the graph screen.• Custom Menus — allows the user to choose among different options.	<ul style="list-style-type: none">• Graphics — different ways to make graphics and sprites.• Compression — if you wanted, you could get into some heavy theory with this.• Self-Modifying Code — code that changes itself while it is executing.• Subprograms — calling one program from another, including external and internal programs.• Assembly — you can make much better programs, but they are larger and more complicated.

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/using-this-guide>

Frequently Asked Questions (FAQ)

This FAQ is an attempt to answer the common TI-Basic related questions that people ask. Many of the questions are related to each other, so it is recommended that you read through the whole list. If you have any questions that aren't mentioned on the list, please post them in the [forum](#) or leave a comment at the bottom of the page.

General

Q: Is TI-Basic easy to learn?

A: Yes! TI-Basic has the majority of the standard features and functionality that you find in other BASIC programming language variants (i.e., things like user input and variables are very similar), so if you can learn those languages, TI-Basic should be no problem. If TI-Basic is your first exposure to programming, it will require some work to learn, but it is definitely worth it because TI-Basic is a fun language to use.

Q: How do I learn TI-Basic?

A: The best way to learn TI-Basic is to download a copy of the manual, and start making small, sample programs to try out the different TI-Basic [commands](#). Once you feel comfortable with the commands, you can start putting them together to create larger programs. After that, you should move on to learning the more advanced design concepts and techniques that are part of TI-Basic.

Q: Where can I get information on TI-Basic?

A: The wiki you are currently on has the largest collection of TI-Basic information available, including [commands](#), design concepts, techniques, and experimentation. The [downloads](#) page has a comprehensive list of TI-Basic tutorials from elsewhere on the Internet, as well as some of the different tools and utilities.

Q: Do you have a tutorial about [subject]?

A: The best way to find out is to use the [search box](#). If you don't find what you are looking for, leave a comment in the [forum](#) and one of us will try to help you. We won't guarantee that you will find everything on this wiki that you are looking for, since it is a constant work in progress and there are simply too many topics to cover. If you would like to make a suggestion for a new tutorial, you can add it to the [wiki to-do list](#).

Q: Some of the tutorials appear to be unfinished. Why is this?

A: Since this is a wiki, and anyone can contribute, our policy is that we will post any legitimate tutorial that contains some useful information. Even if the person who started the tutorial doesn't finish it, there is a good likelihood that someone else will stumble upon it, and can improve or add on to it until it is finished. If you ever see a tutorial that could be improved in some way, we encourage you to just go ahead and change it.

Q: Where did the TI-Basic name come from?

A: Back when the language was growing in popularity and use, people wanted a simple name to refer to it that was easy to remember and told you what it was. Because it is the built-in programming language of the *TI* graphing calculators, and it is a variant of *BASIC*, TI-Basic is

what they called it. You should note that the name is unofficial, as TI has never actually given it a name (for example, try searching for TI-Basic in the calculator manual; you won't find it).

Q: I've seen TI-Basic spelled with all uppercase (TI-BASIC) and with mixed case (TI-Basic), but what is the correct way to spell it?

A: Truthfully, there is no one correct way to spell it. It is just a personal preference. On this wiki, however, you will probably notice that we spell TI-Basic with mixed case. The primary reason for that decision is because it is easier to read (all caps aren't very reader-friendly).

Q: What languages do the TI calculators support?

A: The TI-83+ and TI-84+ support several different languages, including: French, German, Italian, Spanish, Portuguese, Dutch, Danish, Finnish, Norwegian, Swedish, Hungarian, and Polish. You just need to download a language localization application to install the respective language on your calculator. The applications can be found on the [application](#) page on TI's site.

Q: What calculators support TI-Basic?

A: All of the TI graphing calculators have TI-Basic support built-in. Of course, the calculators each have their own TI-Basic variant (see next question).

Q: What's the difference between [TI-83 Basic](#) and [68K TI-Basic](#)?

A: Simply put, a whole lot. TI-83 Basic lacks all sorts of things that 68K TI-Basic has, including indirection, local variables and functions, advanced picture manipulation, text in matrices, and so on. It's a shame, too, because these things are extremely useful, and make TI-Basic that much richer of a language.

Q: Is there a place where I can interact with other TI-Basic programmers?

A: Yes. There is a fairly active [forum](#) available on this site where you can ask questions, get program feedback, share ideas, or whatever else you want to talk about. Another active forum is [United-TI](#), and several of the members of this wiki are active members there.

Games

Q: Where can I find TI-Basic games and programs to download?

A: While there are several calculator sites on the Internet that have TI-Basic programs, including [ticalc.org](#), [calc.org](#), [calcgames.org](#), and [unitedti.org](#), most of the TI-Basic programs are of subpar quality, featuring crappy coding, gameplay, and graphics. If you are looking for quality TI-Basic programs, your best bet is to check out the programs on the [showcases](#) page.

Q: I don't want to buy the Graph Link cable. Can't I just type in the games by hand?

A: Yes, you can type in the games. All you need to do is download the [Graph Link](#) or [TI-Connect](#) software created by TI. You then start up the program, and open your game in the editor. If you don't like the idea of downloading an application, an alternative option is to view the games online using the [SourceCoder](#) application.

Q: I have a Graph Link cable, and want to send a game to my calculator. How do I do that?

A: Assuming you already have either the Graph Link or TI-Connect software (see previous question), you simply start up the software, click send to open the send menu, find your desired game, and then click transfer to send the game to your calculator. Note that if a game has several files that go with it (such as [pictures](#) and [subprograms](#)), you need to send those files with the game in order for it to work correctly.

Q: What is an emulator?

A: An emulator allows you to run a virtual form of your calculator on your computer, which is very convenient when you want to make quick changes to programs, or do any debugging or optimizing. There are several [emulators](#) available for you to use, so you should just experiment to see which one you prefer.

Q: I downloaded an emulator for my calculator, but it won't work because it says it needs a

ROM image. What is that?

A: A ROM image is simply an instance of your calculator, which tells the emulator that you own your calculator. It is primarily used as a safeguard because only one person is supposed to be using any one ROM image. To download the ROM image to your computer, you just link your calculator to your computer, and then the emulator should be able to download the ROM image off of it.

Q: I have an awesome idea for a game, but I don't know how to program. Can you program it for me?

A: While we would like to help you program your game, we each have our own projects that we're working on and other real-world things (like school and a job) that occupy our time, so we aren't able to program your game for you. At the same time, if you have a specific TI-Basic programming question that you need help with, we'd be happy to help you. Even better than us programming your game, though, is you programming it yourself (see next question).

Q: What do I need to make games?

A: The main things you need to make games are your TI [calculator](#) and calculator [manual](#). Before you actually implement a game, however, you should [plan](#) it out. This involves coming up with the idea for the game, and working out the many details of the game: graphics, gameplay, menus, and so on. Once you have all of those things figured out, you just need to put them into action.

Q: What is a good tutorial for making games?

A: Unfortunately, there really is no comprehensive game tutorial available. Instead, there are several small [tutorials](#) that each cover different aspects of games. In addition, on this wiki there are quite a few techniques covered, including [animation](#), [custom menus](#), [saving](#), [highscores](#), [maps](#), and [movement](#).

Q: Can I use a routine from this wiki in my game?

A: Yes! In fact, we encourage it. All of the routines on this site are designed to be as optimized and efficient as possible, so that readers learn the best way to program.

Q: Can I use sprites from other games in my own game?

A: The general consensus among the calculator programming community is that using somebody else's graphics in your game is fine, as long as you get their permission to do so. However, if you don't plan on releasing your game to the community, but instead just keeping it to yourself and your friends, then it doesn't really matter.

Programming

Q: How do I draw graphics?

A: You need to use the [graph screen](#) commands to draw [graphics](#). There are several commands available, including points, pixels, lines, circles, and text. The one caveat you need to be aware of when drawing graphics is that the graph screen settings affect how some of the commands show up. See the respective [command](#) pages for more information.

Q: I've tried using the graphics commands, but they are too slow for my game. Is there a way to get better graphics?

A: In fact, there is. You can use one of the [assembly libraries](#) that is available. In particular, the best two assembly libraries for graphics are [Omnicalc](#) and [xLIB](#). You can use them to create complex sprites, or any of the other advanced graphics that you see in TI-Basic programs.

Q: Can I do [task] in TI-Basic?

A: While it's possible to do almost anything in TI-Basic, whether it looks nice and runs at a decent speed is a different matter. If you have thoroughly [planned](#) your program and made it as [optimized](#) as possible, and your program still takes a minute to load and there's a five second lag after each key press, that's a good indicator that you should probably use Assembly instead. At the same time, you should always strive to push the boundaries of TI-Basic.

Q: How do I convert a number to a string and vice versa?

A: Converting a string to a number is actually very easy, and involves simply using the expr(command. Going the other way, however, is much more complicated because there is no built-in command to do it. What you need to use instead is a small number-to-string routine that involves using the LinReg(ax+b) command in an unorthodox way.

Q: What's the difference between setting a variable to zero and using the DelVar command?

A: When you set a variable to zero, you simply make its value zero. When you use the DelVar command on a variable, you actually delete the variable from memory. For letter variables, the next time the variable is used it is set to zero. DelVar also has some optimization capabilities associated with using it.

Q: How do I un/archive programs from within a program?

A: While the Archive and UnArchive commands would seem like the right commands to use, they actually don't work with programs from inside the program editor — interestingly enough, though, they do work with programs on the home screen outside of the program editor. What you need to use instead is an assembly program, such as Celtic.

Q: I want my program to be run when a person turns on their calculator. Is there a way to do that?

A: Not in TI-Basic, but you can use a Flash application to do that. TI created a Start-Up Customization application which will allow you to run a specific program, application or show a picture on the calculator screen each time the calculator is turned on.

Q: My program is extremely large. Is there a way to manage/condense the code better?

A: Subprograms and optimization are your friends :D

Q: Are there any undocumented features (Easter eggs) in TI-Basic?

A: Of course. Probably the most well-known undocumented feature is large text on the graph screen, which is achieved by placing a -1 at the beginning of the Text(command. Another cool undocumented feature which was recently discovered is the ability to draw circles significantly faster by placing a list with an imaginary *i* after the last argument. Besides those two Easter eggs, the TI-Basic community has made great strides in understanding TI-Basic and its many different facets.

Q: How do you disable the ON key?

A: Unfortunately, you can't. You need to use assembly to disable the ON key.

Q: How do I hide the code of my TI-Basic program?

A: While you can edit-lock a program and employ some other protection mechanisms, that only really prevents novice calculator users from getting access to your code. Anybody who knows what they are doing will have no problem bypassing your program protection.

Q: Where do you get the lowercase letters?

A: Lowercase letters aren't available by default, so you need to use an assembly program to turn on the lowercase flag that the calculator uses for enabling lowercase letters. You then just press Alpha twice to switch to lowercase mode. A good substitute for lowercase letters is the statistics variables, accessible by pressing VARS and then scrolling down to Statistics. Please note that while lowercase letters look nice, they each take up two bytes of memory, instead of the one byte that uppercase letters use.

Troubleshooting

Q: I think some of the routines on this wiki have errors in them, because they didn't work for me. Could you please correct them?

A: We have strived to make sure that all of the routines on this site work correctly and without problems. However, if you are 100% sure that you entered the routine correctly into your

calculator, please leave a comment on the page using the comment function at the bottom of the page. Somebody will then be able to correct the routine so that it won't cause anybody else any problems.

Q: I downloaded a program from the Internet, but it has a .8xg extension instead of the typical .8xp extension. What is that, and how do I get it work?

A: The .8xg means that it is a group file (compared to the .8xp for programs), which you must ungroup in order to use. A group file contains one or more files, which can consist of whatever you want (programs, pictures, variables, or whatever else). The main reason that people group their programs is so that all of the program files are in one file, rather than having to remember lots of separate files; it's a matter of convenience.

Q: I found a TI-Basic program on the Internet, and typed it into my calculator. How come the program doesn't work?

A: While TI-Basic commands and functions look like they are made up of individual characters that you can type in, they are actually tokens that must be obtained by going to the relevant menu or pressing a key. Depending on the size of the program, it might be better to simply download the entire program to your calculator, instead of manually entering it in.

Q: I was playing a TI-Basic game and my calculator suddenly shut off. When I turned it back on, my memory was erased. What happened?

A: Your game had a glitch of some kind, and it caused the calculator to crash. This is usually caused by Assembly programs, as the majority of TI-Basic errors are caught by the calculator. You don't have to worry very much about TI-Basic crashes because they don't do any real permanent damage to the calculator, but because it is very annoying to have to replace all of your programs after your RAM is cleared, you should always store any important files in the archive.

Q: When I tried to run my TI-Basic program, I got this error message. What does it mean?

A: TI-Basic has a built-in error menu, which displays a respective error message based on the error that occurred. If the program is not edit-locked, then it will have a Goto option, which will take you to the point in the code where the error is. There's actually a whole list of error messages that you can receive at any one time, so you should just go down the list and try to see what you did to cause the error.

Q: I downloaded a TI-83 TI-Basic program, and tried to run it on my TI-84+SE calculator. It looked like it should work, but it doesn't. How come?

A: The majority of TI-83 TI-Basic programs will work on the TI-83+ and TI-84+ calculators, and likewise the majority of TI-83+ and TI-84+ TI-Basic programs will work on the TI-83. However, if a program uses either Assembly (Assembly must be compiled for the specific calculator in order to work) or any of the new TI-Basic commands/functions that TI added, then the program will not work on another calculator. See program portability for more information.

Q: After I finished running a TI-Basic game, my screen was split in two between the home screen and the graph screen. On top of that, the axes on the graph screen were gone. How do I get my calculator back to normal?

A: Unfortunately for you, the person who programmed the game didn't do a good job of cleaning up after their program, so you have to do that yourself. The calculator has several different settings that you can change to make it look however you want. In this case, you want the screen to just show the full home screen, so the appropriate command would be Full. Turning the axes back on can be accomplished by using the AxesOn command.

Assembly

Q: What is assembly?

A: Assembly is the other primary programming language available for the TI-83 series of calculators, and it is a low-level language programmed in the calculator's own machine language.

Q: How does TI-Basic compare to assembly?

A: TI-Basic is much easier to learn and program in, but it is rather slow because it is an interpreted language. This not only affects getting user input, but also displaying text and graphics on the screen. Assembly on the other hand, is much harder to learn, but allows you to make all sorts of complex games and programs that look nice and run at a decent speed.

Q: Is it possible to convert TI-Basic to assembly?

A: No, it is not. There are currently no working programs available that will convert TI-Basic to assembly (note: I say working because people have tried creating TI-Basic to assembly converters, but nobody has completed one yet), so the only way you can convert a TI-Basic program to assembly is by learning assembly and porting the program yourself. You could also try asking an assembly programmer to port it for you, but most people won't do that unless the program is pretty small.

Q: I want to use an assembly program with my TI-Basic program, but I can't figure out how to use it. Can you help me?

A: Unfortunately, we really can't do much for you. What we recommend is that you contact the author of the assembly program and ask them for help. They wrote the program, so naturally they should be able to answer any questions that you have.

Q: When I tried to run a program from the program menu, it gave me a ERR:SYNTAX error with no Goto option. Why would it do that?

A: This means that you tried to run an assembly program. Most assembly programs are run from an assembly shell, such as MirageOS or Ion, but there are some assembly programs that you can run just using the Asm(command. These are commonly called nostub.

Q: Why would I want to run my TI-Basic program from an assembly shell?

A: Most people have assembly games on their calculator, which require an assembly shell to run, so they get accustomed to running their programs through a shell. In addition, they like being able to run all their games in a shell, including their TI-Basic games, so they don't have to exit the shell. Truthfully, though, there is really no advantage to running a TI-Basic program from a shell. It's just a personal preference.

Q: How do I run my TI-Basic program from an assembly shell?

A: The standard way to get a TI-Basic program to appear in an assembly shell is to add a special header to the beginning of the program. This header consists of a colon (:), and then you can add an optional program description that will be displayed together with your program in the shell.

```
PROGRAM: SAMPLE
```

```
::"Program name // note the two colons
:// program code
```

DoorsCS also has support for a custom icon, which needs to be stored in hexadecimal format. You should look at the documentation for DoorsCS to see how to use it.

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/faq>

Overview of Commands

The TI-Basic programming language features a wide range of commands for all kinds of things, including output, user input, manipulating variables, linking calculators, and especially math —

don't forget a calculator is primarily designed for doing math. In total, there are almost 400 commands on the TI-83+ calculators, and the TI-84+/SE calculators have an additional 22 commands (because of the new time and date commands and the new math commands added in OS 2.30).

Because sorting through all of these commands is a rather daunting task, especially when trying to figure out which one is appropriate for accomplishing a desired task, we have chosen to present them in three different formats:

- **Alphabetical Index** — A listing of the commands sorted in alphabetical order.
- **Menu Map** — A listing of the commands sorted by where they appear in their different menus on the calculator.
- **Category** — A listing of the related commands grouped together based on their common function and purpose.

There are eleven different command categories that we have come up with:

- | | | |
|---|--|--|
| <ul style="list-style-type: none">• <u>Home Screen</u>• <u>Graph Screen</u>• <u>Math Functions</u>• <u>Variables</u> | <ul style="list-style-type: none">• <u>User Input</u>• <u>Operators</u>• <u>Calculator Linking</u>• <u>Controlling Flow</u> | <ul style="list-style-type: none">• <u>User Settings</u>• <u>Memory Management</u>• <u>Time and Date</u> |
|---|--|--|

We tried to do our best to come up with these categories because we believe they reflect the natural divisions of the commands, although some of the commands have functionality that overlaps more than one category. In this case, we decided to simply go with the primary use of the command.

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/commands>

The Home Screen and Its Commands

The TI-83+/SE home screen is composed of eight rows (1 to 8 from top to bottom) by sixteen columns (1 to 16 from left to right); it is like a grid. The home screen uses the large, easy to see, 5 by 7 font. Because each character takes up the same 5 by 7 space, regardless of what its actual size is, the text cannot be moved around to get pixel perfect precision.

The table below shows the coordinates used for the Output(command (see below). Enter these coordinates in the Output(command as shown, only without the parentheses.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)	(1,8)	(1,9)	(1,10)	(1,11)	(1,12)	(1,13)	(1,14)	(1,15)	(1,16)
2	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)	(2,8)	(2,9)	(2,10)	(2,11)	(2,12)	(2,13)	(2,14)	(2,15)	(2,16)
3	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)	(3,8)	(3,9)	(3,10)	(3,11)	(3,12)	(3,13)	(3,14)	(3,15)	(3,16)
4	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)	(4,8)	(4,9)	(4,10)	(4,11)	(4,12)	(4,13)	(4,14)	(4,15)	(4,16)
5	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)	(5,8)	(5,9)	(5,10)	(5,11)	(5,12)	(5,13)	(5,14)	(5,15)	(5,16)
6	(6,1)	(6,2)	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)	(6,8)	(6,9)	(6,10)	(6,11)	(6,12)	(6,13)	(6,14)	(6,15)	(6,16)
7	(7,1)	(7,2)	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)	(7,8)	(7,9)	(7,10)	(7,11)	(7,12)	(7,13)	(7,14)	(7,15)	(7,16)

The home screen does not have access to any of the drawing commands that are available on the graph screen (such as the points, pixels, lines, or circles). This leaves you with just using the text to imitate graphics, which unfortunately does not look very good. Using the home screen is faster than using the graph screen, though.

There are five main home screen commands:

- **ClrHome** — Clears the home screen of any text or numbers. It should be used at the beginning of a program and at the end to make sure the user has a clear screen afterwards.
- **Disp** — Displays one or more arguments of text or values on a new line on the home screen and scrolls down if necessary. Disp should be used instead of Output(in most cases.
- **Output(** — Displays text or a value at a specified row and column location on the home screen. It also wraps the text or value around the screen if needed.
- **Pause** — Pauses the program and displays the home screen until the user presses ENTER. It also can display text or a value with scrolling available.
- **Menu(** — Displays a generic menu on the home screen, with up to seven options for the user to select from. It utilizes branching to make the menu.

You should commit yourself to learning how to use these commands and then actually start using them in your programs. They are rather basic, but still quite powerful. Once you have them down, move on to the graph screen commands.

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/homescreen>

The Graphscreen and Its Commands

The TI-83+/SE graphscreen is 64 rows, by 96 columns, the coordinates for the pixels being 0 to 62 a total of 63 and 0 to 93 a total of 94. So amount of editable X and Y pixels are 95 and 63. With the coordinates 0,0 actually being the very top left pixel. The graphscreen uses the small (3 by 5) font, which allows you to display more text; and the large (5 by 7) font, which allows you to make the graphscreen look like the homescreen. Graphics can also be displayed on the graphscreen, in the form of points, pixels, lines, or circles, as well as shading an area of the graphscreen. These graphics can be displayed with the text, and they can be saved to pictures for later use.

The graphscreen does not have access to some of the commands that are available on the homescreen (such as the user input). In addition, some of the graphscreen commands have their coordinates reversed, so the row comes before the column. These commands also take longer to draw.

fold

Table of Contents

- [Setting up the Graphscreen](#)
- [Clearing the Graphscreen](#)
- [Adjusting the Window Dimensions](#)
- [Turning off the Graph Formats](#)

[Graph Databases \(GDB\)](#)
[Graphing Functions on the Graphscreen](#)
[Displaying Text on the Graphscreen](#)
[Drawing & Shading on the Graphscreen](#)
[Drawing Points](#)
[Drawing Pixels](#)
[Drawing Lines](#)
[Drawing Circles](#)
[Shading Areas](#)
[Storing the Graphscreen to a Picture](#)
[Advantages & Disadvantages of Pictures](#)

Setting up the Graphscreen

Before using the graphscreen, you first need to set it up appropriately. When displaying text or drawing graphics, you want to make sure that they are displayed how you want them to be displayed. This is achieved by clearing the graphscreen, adjusting the window dimensions, and turning off the graph formats. You should also store the current graphscreen settings into a graph database.

Clearing the Graphscreen

The ClrDraw command is the graphscreen equivalent to ClrHome. ClrDraw is usually used before you display text or draw anything on the graphscreen, to ensure that it won't be interrupted by anything that was previously displayed on the graphscreen.

Format
:ClrDraw

You also want to make sure to clear the graphscreen when exiting programs. This ensures that the next program that the user runs won't have to deal with whatever text or graphics your program left behind. It also helps the user, because they won't have to manually clear the graphscreen.

To use the ClrDraw command, you should first be in the Program editor for your program. In the Program editor, press 2nd and PRGM. Then press ENTER on ClrDraw. Now the command has been put into your program.

Adjusting the Window Dimensions

After clearing the graphscreen, you will want to set the window dimensions to the desired size. There are four window variables that control the window dimensions: Xmin, Xmax, Ymin, and Ymax. When storing values in these variables, you have to remember that the max variables always have to be larger than the min variables, otherwise you will get an error.

The Xmin variable sets the minimum value that the X coordinate can have. The Xmax variable sets the maximum value that the X coordinate can have. The Ymin variable sets the minimum value that the Y coordinate can have. The Ymax variable sets the maximum value that the Y coordinate can have. You can use these variables like regular variables.

Format
:#→Xmin:#→Xmax

```
:#→Ymin:#→Ymax
```

Although the graphscreen is 96 pixels wide by 64 pixels tall, the bottom row is unusable in TI-Basic programs and the far right column is reserved for the pause indicator. So, most people set the window dimensions to 0 for Xmin and Ymin, 94 for Xmax, and 62 for Ymax. This sets the X range from -1 to 94 totaling 95 columns and the Y range from -1 to 62 totaling 63.

Format

```
:0→Xmin:94→Xmax  
:0→Ymin:62→Ymax
```

In a simpler notion, to make everything positive from the bottom left corner you would use the following code.

Format

```
:1→Xmin:95→Xmax  
:1→Ymin:63→Ymax
```

Others set 0 for Xmin and Ymax, 94 for Xmax and -62 for Ymin. This allows them to use the same coordinates for pixel and point commands, as pixel rows on the screen are counted starting from the top. So the top pixel row is 0 and the bottom row is 62, while the point top row is 0 and bottom row is -62.

In addition to those four window variables, there are two other window variables that you can use to set the window dimensions. ΔX is the difference between Xmin and Xmax divided by the graphscreen width and ΔY is the difference between Ymin and Ymax divided by the graphscreen height. When you set Xmin and Ymin to 0, you just need to set ΔX and ΔY to 1 to make Xmax 94 and Ymin 62.

```
:0→Xmin:94→Xmax  
:0→Ymin:62→Ymax
```

Replace with ΔX and ΔY

```
:0→Xmin:1→ΔX  
:0→Ymin:1→ΔY
```

To use the window variables, you should first be in the Program editor for your program. In the Program editor, press VARS and 1, then scroll down to whichever variable you want to use and press ENTER. Now the variable has been put into your program. You then have to type what number you want to set the window variable to.

In addition to setting the window variables individually, there are also a couple commands that can set them all at the same time. Although these commands are only useful in a couple situations, they are a lot easier (and smaller) to use.

The ZStandard command sets the window dimensions to their default settings (which is -10 for Xmin and Ymin, and 10 for Xmax and Ymax). The ZSquare command sets the window dimensions so that they make the screen square. This is important when drawing circles because it makes the circles look like circles (instead of ellipses).

Format

```
:ZStandard
```

:ZSquare

To use the ZStandard or ZSquare command, you should first be in the Program editor for your program. In the Program editor, press the ZOOM key. Then scroll down to whichever command you want to use and press ENTER. Now the command has been put into your program.

Turning off the Graph Formats

After adjusting the window dimensions, you will want to turn off the graph formats. The graph formats include the grids, plots, axes, and functions. These can be turned off and turned on, depending on what is desired.

The GridOff command turns the grid off and the GridOn command turns the grid on. The PlotsOff command turns the plots off and the PlotsOn command turns the plots on. The AxesOff command turns the axes off and the AxesOn command turns the axes on. The FnOff command turns all of the functions off and the FnOn command turns all of the functions on.

```
Format  
:GridOff/On  
:PlotsOff/On  
:AxesOff/On  
:FnOff/On
```

The plots and functions commands can also be used to just deal with one or two plots or functions, instead of all of them. You just put the plots or functions and their numbers after the command, separating each one with a comma.

```
Format  
:PlotsOff/On function#[,function#,...]  
:FnOff/On function#[,function#,...]
```

To use the graph formats, you should first be in the Program editor for your program. In the Program editor, press 2nd and ZOOM. Then scroll down to GridOff/On or AxesOff/On and press ENTER. To use the PlotsOff/On or FnOff/on commands, you need to press 2nd and 0 for the Catalog menu. Then scroll down to the command or press the first letter of the command and press ENTER. Now the command has been put into your program.

Graph Databases (GDB)

There are 10 graph database (GDB) variables (GDB0 through GDB9) that store the window and graph format settings, so they can later be used to recreate the graphscreen; GDBs do not contain graphics or stat plot definitions. If a program utilizes the graphscreen, it should restore the graphscreen settings with a GDB when the program finishes executing.

The StoreGDB command saves the graph settings in a GDB. It is best used at the beginning of a program. The RecallGDB command restores the graph settings that are stored in a GDB. It is best used after a program is finished executing. You should remember to delete the GDB after recalling it.

```
Format  
:StoreGDB #
```

```
:RecallGDB #
```

To use the GDB commands, you should first be in the Program editor for your program. In the Program editor, press 2nd DRAW and > > twice. Then scroll down to StoreGDB or RecallGDB and press ENTER. Finally, press the number of the GDB you want to use.

Putting all of these commands together, here is a typical way to set up the graphscreen at the beginning of a program:

```
PROGRAM:GRAPHSET
:StoreGDB 1
:ClrDraw
:GridOff
:PlotsOff
:AxesOff
:FnOff
:0→Xmin:1→ΔX
:0→Ymin:1→ΔY
```

Graphing Functions on the Graphscreen

Graphing functions is primarily used in math programs. There are three commands that are used for graphing functions: DrawF, DrawInv, and Tangent. The commands do not change the function variables, and their graphs (and tangent line) are erased when any command changes the graphscreen.

The DrawF command graphs an expression. The DrawInv command graphs the inverse of an expression by plotting X values on the y-axis and Y values on the x-axis (as if the X and Y values are switched). The Tangent command draws a line tangent to the expression, with the line touching the expression at the X value. Use the Input or Pause command to view the graph (or tangent line).

```
Format
:DrawF expression
:DrawInv expression
:Tangent(expression,value)
```

The expression can either be a function variable (Y0 through Y9) or a function in terms of X (such as $3X+4$). While you create a function variable by storing an expression (enclosed in quotes) to it, the function in terms of X is just the expression itself (which allows you to bypass the function variable). The expression can consist of numbers, variables, and math functions.

```
Format
:"expression"→Y#
:expression
```

After the function variable is created, it is stored in the Y= editor and selected to be graphed. If you already have an expression stored in a function variable, you can combine function variables with other expressions to create new expressions. You cannot use a list in the expression to draw several graphs at one time.

When graphing functions, you have to adjust the graph formats and window dimensions to ensure the functions display correctly on the screen. Although you can have successive graphs (graphs displayed on top of each other), this is sometimes unwanted because it interrupts the graphscreen while you're graphing your functions. You can get rid of this problem by using the FnOff command.

To use the graph commands, you should first be in the Program editor for your program. In the Program editor, press 2nd and PRGM, then scroll down to whichever command you want and press ENTER. Now the command has been put into your program. You then have to type the expression you want to graph. The function variables can be found by pressing VARS, pressing > once to get to the Y-Vars menu, and then scrolling down to Functions.

Displaying Text on the Graphscreen

The Text command displays text, numbers, variables, or expressions wherever you want on the graphscreen. Because the Text command utilizes the small font (available only on the graphscreen), more text can be displayed on the screen. The Text command overwrites any existing text on the screen, and it is also not affected by the graphscreen window settings.

When you use the Text command, you need to specify the starting coordinates of what you want to display. You first specify the row (0 to 57 from top to bottom) and then the column (0 to 91 from left to right). Although the graphscreen is actually 94 rows by 62 columns, you will get an error if you try to display text on a higher row or column.

The reason is that the graphscreen text is five pixels tall and a variable width. While numbers, uppercase letters, and most lowercase letters are three pixels wide, some lowercase letters (such as w and m) are five pixels wide, and spaces are one pixel wide. There is an automatic space (one pixel wide) inserted between text. So, you need to factor in the height and width of the characters when positioning them on the screen.

```
Format  
:Text(row,col,argument)
```

A good way to find where exactly you want to place the text or other drawing is to use a blank Input command. This gives you a cursor to find where to put it. Just remember that the coordinates at the bottom of the screen are not what you put into the Text(command.

```
Format  
:Input
```

The Text command can display multiple arguments of both text and variables on the same line, at the same time. This is very useful because it eliminates the need to have to worry about spacing. If the variable changes, the Text command will adjust it on the screen accordingly. This allows you to sometimes remove multiple Text commands and just use the first one to display everything.

```
:Text(5,5,A  
:Text(5,9,"/  
:Text(5,13,B  
Combine Text Commands  
:Text(5,5,A,"/",B
```

On the TI-83+/SE calculators, the Text command can also display the large font that is available on the homescreen. Just put a negative one (-1) before the row and column arguments. When using the large font, you have to keep formatting in mind because it is very easy for the text to go off the screen. This is useful when you want to clear large portions of the graphscreen at a time.

Format

```
:Text(-1, row, col, argument)
```

If you have a string of numbers that you are displaying, you don't need to put quotes around the numbers. You may want to keep the numbers in a string, though, if they have any leading zeros. Because the numbers are no longer in a string, the leading zeros will be truncated (taken off) and not be shown.

```
:Text(2, 2, "2345  
Remove the Quotes  
:Text(2, 2, 2345
```

To use the Text command, you should first be in the Program editor for your program. In the Program editor, press 2nd and PRGM. Then, scroll down (all the way at the bottom) to Text and press ENTER. Now the Text command has been put into your program. You can then begin typing some text by turning on the alpha-lock with pressing 2nd and ALPHA.

Drawing & Shading on the Graphscreen

Drawing on the graphscreen is one of the main uses of the graphscreen. There are several different things that you can draw, including points, pixels, lines, and circles. Besides drawing, you can also shade in an area on the graphscreen with whatever size and look you want.

Drawing Points

The point commands are used to draw points on the graphscreen. A point is just a pixel on the screen. The point commands use the (x,y) coordinate system, which is affected by the window settings. This means you have to change the window settings accordingly when you use the point commands, otherwise the points won't show up correctly.

The Pt-On command turns on the point at the given (x,y) coordinates. The Pt-Off command turns off the point at the given (x,y) coordinates. The Pt-Change command toggles the point at the given (x,y) coordinates. If the point is on, it will be turned off and vice versa.

Format

```
:Pt-On(x, y)  
:Pt-Off(x, y)  
:Pt-Change(x, y)
```

The Pt-On and Pt-Off commands also have an optional mark argument that determines the shape of the point. The mark can be either one (dot), two (3x3 box), or three (3x3 cross). You don't need to specify the mark when using the first mark because it is the default. Remember to use the same mark when turning a point off as you used to turn it on.

```
:Pt-On(5,5,1  
Remove Mark  
:Pt-On(5,5
```

To use the point commands, you should first be in the Program editor for your program. In the Program editor, press 2nd and PRGM, then press right once and scroll down to whichever command you want and press ENTER. Now the command has been put into your program. You then have to type the numbers for where you want the point to appear on the screen.

Drawing Pixels

The pixel commands are the alternative way to draw pixels on the graphscreen. Although they are easier to use because they are not affected by the window settings (which means you don't have to set the window dimensions when using them), the coordinate system is switched around so that the row comes first and then the column — it's (y,x) instead of (x,y).

The Pxl-On command turns on the pixel at the given (y,x) coordinates. The Pxl-Off command turns off the pixel at the given (y,x) coordinates. The Pxl-Change command toggles the pixel at the given (y,x) coordinates. If the pixel is on, it will be turned off and vice versa. The pixel commands are faster than their equivalent point commands, so they should generally be used instead whenever possible.

```
Format  
:Pxl-On(y,x)  
:Pxl-Off(y,x)  
:Pxl-Change(y,x)
```

Besides these three commands that have point equivalents, there is also a Pxl-Test command. The Pxl-Test command checks whether the pixel at the given (y,x) coordinates is on or off. One is returned if the pixel is on and zero is returned if the pixel is off. You can store the result to a variable for later use, or use the command in a conditional or loop.

```
Format  
:Pxl-Test(y,x)
```

To use the pixel commands, you should first be in the Program editor for your program. In the Program editor, press 2nd and PRGM, then press right once and scroll down to whichever command you want and press ENTER. Now the command has been put into your program. You then have to type the numbers for where you want the pixel to appear on the screen.

Drawing Lines

The Line command allows you to draw a line anywhere on the screen. The line can be any length that you want. When using the Line command you need to supply the coordinates of the two endpoints. The Line command draws the line from the first endpoint (x₁,y₁) to the second endpoint (x₂,y₂).

```
Format  
:Line(x1,y1,x2,y2)
```

The Line command has an optional fifth argument that controls whether the line will be drawn

(the argument should be one) or erased (the argument should be zero). The line is drawn by default, so it should be left off unless you want to erase it.

```
:Line(5,5,10,5,1  
Remove Line's Fifth Argument  
:Line(5,5,10,5)
```

When you have multiple pixels in a straight line that you turn on or off, you can sometimes replace the pixel commands with one or more Line commands. In the case that the pixels are arranged at a slant or angle, you can just adjust the line coordinates accordingly. You should also use Line commands instead of pixel commands when clearing large portions of the graphscreen at a time.

```
:Px1-On(5,5  
:Px1-On(5,6  
:Px1-On(5,7  
Replace with Lines  
:Line(5,5,5,7)
```

There are two other line commands that are also available. They are primarily designed for when you want to quickly draw a line across the entire screen. The Horizontal command draws a horizontal line at a given row and the Vertical command draws a vertical line at a given column. The argument can either be a number or a variable.

```
Format  
:Horizontal y  
:Vertical x
```

To use the line commands, you should first be in the Program editor for your program. In the Program editor, press 2nd and PRGM, then scroll down to whichever command you want and press ENTER. Now the command has been put into your program. You then have to type the number(s) for where you want the line to appear on the screen.

Drawing Circles

The Circle command draws a circle on the graphscreen. When using the Circle command, you must enter three numbers (separated by commas): the (x,y) coordinates of the center of the circle and the length of the radius. Because circles take a long time to draw, you should use them sparingly.

```
Format  
:Circle(x,y,radius)
```

The window settings affect how the circles are drawn. With the screen height being larger than the width (the screen is a rectangle), the circles will actually look like ellipses. To make them look like circles, you need to use the ZSquare command.

To use the Circle command, you should first be in the Program editor for your program. In the Program editor, press 2nd and PRGM, then scroll down to Circle and press ENTER. Now the Circle command has been put into your program. You then have to type the numbers for where

you want the center of the circle to be and the length of the radius.

Shading Areas

The Shade command shades in an area on the graphscreen. For basic shading, you just need to specify a lower function and upper function. The Shade command will vertically shade in the area that is above the lower function and below the upper function across the whole length of the screen. The functions can either be a function variable (Y0 through Y9); or a function in terms of X, consisting of numbers, variables, and math functions (such as $3X+4$).

Format

```
: Shade(lowerfunction,upperfunction)
```

When shading with the function variables, the window settings affect how the shading looks. You should set the window variables to ensure that the shading is done correctly. This also applies when you are shading at the same time as drawing, because some of the drawing commands are affected by the window settings.

Because you might not want to shade the whole length of the screen, the Shade command has two optional arguments that allow you to specify the left and right boundaries for shading (the boundaries themselves are also shaded). Xleft and Xright can be whatever numbers you want, as long as they are between Xmin and Xmax (the horizontal window dimensions). You can also just specify Xleft by itself if you only want to change the left boundary (you need to set Xleft to set Xright, though).

Format

```
: Shade(lowerfunction,upperfunction,Xleft,Xright)
```

The Shade command has two other optional arguments that allow you to change the look of the shading, but you need to also set the left and right boundaries to use them. The pattern can be either one (vertical), two (horizontal), three (slanted backwards), or four (slanted forwards); and the patres (the frequency of the shading) can be from one to eight pixels.

Format

```
: Shade(lowerfunction,upperfunction,Xleft,Xright,pattern,patres)
```

To use the Shade command, you should first be in the Program editor for your program. In the Program editor, press 2nd and PRGM, then scroll down to the command (or press the 7 key) and press ENTER. Now the command has been put into your program. You then have to specify the functions (and boundaries, pattern, and patres) for how you want to shade the area.

Storing the Graphscreen to a Picture

After you have spent lots of time drawing something on the graphscreen, you naturally want to keep it for future use. So, you store it to a picture variable. A picture variable holds a copy of the graphscreen at the respective time it was stored to. Although pictures are often used, it really is a personal preference.

The StorePic command saves the current graphscreen to the designated picture. After saving a picture, you can display it again with the RecallPic command. Before recalling a picture, you should first make sure that the graphscreen is clear. This ensures that the picture won't be

interrupted by anything that is already on the screen. You can only use numbers with the StorePic and RecallPic commands; no variables.

```
Format  
:StorePic #  
:RecallPic #
```

Because each picture takes up 768 bytes, you should delete them when exiting programs. The program should only keep the picture if it is used for something important (such as a titlescreen). The user doesn't want to have their memory cluttered up with lots of variables.

To use the picture commands, you should first be in the Program editor for your program. In the Program editor, press 2nd DRAW and < once. Then scroll down to StorePic or RecallPic and press ENTER. Finally, press the number of the picture you want to use.

Advantages & Disadvantages of Pictures

The main advantage of using pictures is that the graphics show up almost instantly compared to the slow speed of drawing them. This is particularly noticeable the more detailed the graphics are and depending on what graphics (primarily circles) are being drawn. Speed is a top priority in most programs (because the user doesn't want to wait), so pictures are usually used.

The main disadvantage of using pictures is that there are only ten pictures (from Pic0 to Pic9). With every program sharing the pictures, there can be conflict when two programs use the same picture (the picture will usually be overwritten by the other program).

Another disadvantage of using pictures is that it is another file that the user needs in order to use your program. If you give someone your program, you will have to also give them your pictures. Your program won't work properly anymore if somebody deletes your pictures or forgets to include them with your program. Although this is mostly out of your hands, users will think your program is at fault.

Pictures have a weird behavior, that can be a curse or blessing. When recalled, they only turn on pixels — not turn them off. Try recalling one picture, then recalling a different picture. You'll notice they overlap. This could be used to instantly blacken a screen, or turn on certain pixels regardless of the graph screen's state.

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/graphscreen>

Math Functions

Calculators are built with one primary purpose: math. Programming, game playing, and everything else is secondary. Thus, you will find a number of powerful math commands. Although it may seem that they are of no use to a programmer, programs sometimes need math functions, and many math functions can be used in clever ways. In this guide we'll group the commands into the following general groups:

Number Operations

These commands deal with different ways

Complex Number Operations

you can manipulate the integer and fraction parts of a number, and are mostly found in the MATH-NUM menu.

- **►Frac**, **►Dec** — display a number as either a *fraction* or a *decimal*.
- **iPart(**, **int(**, **fPart(**, **round(** — take the integer or fractional part of a number in various ways.

Probability and Combinatorics

These commands are generally found in the MATH-PRB menu (except for **randM(**, which is in the MATRIX-MATH menu). They include commands for generating random numbers, and commands that are useful for solving problems in combinatorics and probability theory.

- **rand**, **randInt(**, **randNorm(**, **randBin(**, **randM(** — pseudorandom number generation.
- **nPr**, **nCr**, **!** — combinatorical quantities.

Calculus

Although the TI-83 series calculators don't, by themselves, have the capability for symbolic calculations, these commands (found in the main MATH menu) can provide numerical approximations for some commonly computed quantities in calculus.

- **fMin(**, **fMax(** — numerical function optimization in one variable.
- **nDeriv(**, **fnInt(** — derivatives and integrals.
- **solve(** — numerical solution of an equation in one variable.

Trigonometry

These commands allow you to manipulate angles, and are generally affected by **Radian mode** and **Degree mode** (so you should check those pages out). Some of these commands live in the 2nd ANGLE menu, some are on the keyboard, and some can only be found in the 2nd CATALOG menu:

These commands are used for dealing with complex numbers, and are found in the MATH-CPX menu. Many other math commands work on complex numbers too, and complex numbers are fairly strongly connected to trigonometry.

- **i** — the basis of complex number theory.
- **conj(**, **real(**, **imag(**, **angle(**, **abs(** — manipulate complex numbers.
- **►Rect**, **►Polar** — display complex numbers in either rectangular or polar form.

Operators

These commands are found on the keyboard and in the 2nd TEST menu, and deal with basic mathematical and logical commands.

- \pm , \div , \times , $\sqrt{}$, \wedge — math operators.
- \equiv , \neq , \leq , \geq , \leqq , \geqq — relational operators.
- **and**, **or**, **xor**, **not(** — logical operators.

Powers, Inverses, Exponentials, and Logarithms

These commands are found all over the place, many on the keyboard itself, and deal with raising some value to a power, or raising a number to some value, or the inverses of those operations.

- \wedge^1 , \wedge^2 , \wedge^3 , $\sqrt[1]{}$, $\sqrt[3]{}$, $\sqrt[x]{}$ — returns the number raised to the respective power.
- **10^a(**, **e^a(**, **E**, **ln(**, **log(** — commands for dealing with exponentials.

Miscellaneous

These commands have nothing to do with each other, but don't really belong to other categories either.

- **π**, **e** — famous (and occasionally, even useful) math constants.
- **min(**, **max(** — returns the smaller number(s) or the larger number(s).

- $\sin()$, $\sin^{-1}()$, $\cos()$, $\cos^{-1}()$, $\tan()$, $\tan^{-1}()$ — trigonometric or circular functions.
- $\sinh()$, $\sinh^{-1}()$, $\cosh()$, $\cosh^{-1}()$, $\tanh()$, $\tanh^{-1}()$ — hyperbolic functions.
- $^\circ$ and r — for converting between various angular units.
- P \blacktriangleright Rx(), P \blacktriangleright Ry(), R \blacktriangleright Pr(), R \blacktriangleright Pθ() — for converting between rectangular and polar coordinates.
- DMS — Formatting command for degrees-minute-second notation.

Comments

Note that the statistics commands are not included here. That is because statistics is not math. There is also the finance commands and variables, which are another topic unto themselves.

In addition, all of the above commands, except for the calculus and random number generating commands, can be applied to lists as well. For single-argument commands, this just means that the command is applied to each element of the list separately.

For multiple-argument commands, there are two ways to do it: with a number and a list (then, the command is applied with that number to each element of a list), and with two lists (then, it's done pairwise, and the lists must be the same size, otherwise the calculator will throw a ERR:DIM MISMATCH error).

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/math>

Variable Types

Variables are used extensively in programming, as most programs use variables in one form or another. They are used to keep track of numbers or text or stats; there are many uses for variables. Put simply, programming wouldn't be programming without variables.

A variable is a reference to the information that it holds. Variables allow you to store the information, so that you can later use it for whatever purpose is desired. The thing to remember, though, is that programs all share the variables.

There are several different kinds of variables available on the calculator, but the four main variables that you will be using are numerics, lists, matrices, and strings. Numerics are used for storing a single number. Lists are used for storing a collection of numbers. Matrices are used for storing numbers in a two-dimensional format. And, strings are used for storing text.

fold

Table of Contents

- Storing & Deleting Variables
- Numeric variables
- List Variables
- Matrix Variables
- String Variables
- Picture Variables and GDBs

[System Variables](#)

[Converting Between Variable Types](#)

[Between lists and matrices](#)

[Between strings and numbers](#)

[Archiving and Unarchiving Variables](#)

Storing & Deleting Variables

Variables have values stored in them so that the values can be recalled for later use. When storing an expression containing a variable into another variable, the value of the variable at that time is used. The store (\rightarrow) command is used for storing variables, and it is accessed by pressing the [STO►] key. When storing a value in a variable, you have the value on the left side of the store command and the variable that it will be stored to on the right side.

Format

:value \rightarrow variable

When you are done using variables, you should delete them with the DelVar command to save space. The DelVar command deletes the contents of a variable from memory. If the DelVar command is used with a real variable, the variable is not only deleted from memory but automatically set to zero the next time it is used. DelVar does not work on specific elements of a list or matrix. In fact, it will actually return an error.

Format

:DelVar variable

Numeric variables

Numeric variables are used for storing numbers. There are 27 numeric variables (from A to Z and θ) that can be easily accessed, and more that the calculator uses for its specific purposes.

Most numeric variables can either be real or complex (the latter involve i , the square root of -1, and are important to advanced algebra). In either case, up to 14 digits of a number can be stored, although only the first 10 will be displayed and used for comparison.

To access a real variable, press ALPHA and then the key corresponding to whatever letter you want your variable to be. You can initialize a real variable by storing a number, another variable, or an expression into the variable using the STO key (or, just using it almost anywhere will initialize it to 0).

List Variables

Lists are used to hold multiple numbers at once, in a specific order. There are six "default" lists named L_1 through L_6 , but an important feature of lists is that they can be given names, so that there are millions of possible lists. Lists are important for programmers for many purposes - saving data after a program finishes running, and storing a level of a game are only two of them.

(for more information, see [Lists and Their Commands](#))

Matrix Variables

Matrices are two-dimensional lists (row by column). Equivalent to lists, they are used when the data needs more structure. Matrices are often used for storing a level or a map of the screen. There are only ten matrices available (from [A] to [J]).

(for more information, see [Matrices and Their Commands](#))

String Variables

Strings are used for storing a sequence of characters, that is, text. A common use for strings is to manipulate text to be displayed in a program, but they have many different purposes: highscores, level and map data, and whatever else is desired. Although there are only ten built-in string variables (Str0 through Str9) available to use, strings can hold many different kinds of characters, including letters (both uppercase and lowercase), numbers, functions, and even other commands. The amount of free RAM is the only limit on the number of characters in a string.

(for more information, see [Strings and Their Commands](#))

Picture Variables and GDBs

Picture variables and GDBs (short for Graph DataBase) are used to save two different elements of the current graph display. A picture variable is used to store the exact appearance of the graph screen. A GDB is used to store system variables relevant to the graph screen - equations, window settings, and the like. 10 built-in variables of each type exist: Pic0 through Pic9 for pictures and GDB0 through GDB9 for GDBs.

(for more information, see [Pictures and GDBs](#))

System Variables

System variables are, for the purposes of this guide, variables that certain commands will use or modify without asking (i.e. without supplying them in the command's arguments). This is a somewhat ill-defined category, and in fact the system variables we'll discuss are of a somewhat miscellaneous nature. They include equation and plot variables, window and table parameters, statistical variables, and finance variables.

(for more information, see [System Variables](#))

Converting Between Variable Types

Between lists and matrices

The [List►matr\(](#) and [Matr►list\(](#) commands are used to convert between a matrix and several lists. Using these commands, it should be simple to implement any kind of conversion between these two data types.

Between strings and numbers

It is very easy to convert a string version of an expression to a number, list, or matrix: the [expr\(](#) command can do it — for example, `expr("5")` will give you the number 5, and `expr("{1,2,3}")` will give you the list {1 2 3}.

Going the other way, however, is slightly more complicated because there is no built-in command to do it. What you need to use instead are a few small routines: see [number to string](#) for how to convert a number to a string. To convert a list or matrix, convert each individual element instead.

Archiving and Unarchiving Variables

On the TI-83+/84+/SE calculators, you can [archive](#) and [unarchive](#) variables. What this entails is the calculator moving the variable to the archive memory or the calculator moving the variable to RAM respectively. The main advantage of archiving a variable is that it is protected from calculator crashes, which clear the calculator's RAM. At the same time, you can't access a variable that's archived; it needs to be in RAM to use it.

```
:Archive L1  
:UnArchive Str1
```

There are a couple things you need to be aware of when using Archive and UnArchive. First, since the TI-83 only has RAM, archiving is not possible, and subsequently neither of these commands are available. This means that you shouldn't use either of these commands if you plan on [porting](#) a program to the TI-83. Second, archiving does not work with the majority of the [system variables](#), including the graphing, statistical, and finance variables. You can archive the other types of variables, however, although list variables are actually more manageable using the [SetUpEditor](#) command.

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/variables>

Getting Input from the User

Getting user input is a basic part of almost all programs. It provides a way of changing variables or transferring control to the user. The four commands used for getting input are: Prompt/Input, getKey, and GetCalc.

User input includes getting values for variables on the calculator, getting the keys that the user pressed, and getting a variable off of or sending a variable to another calculator over a link cable.

fold

Table of Contents

[Getting Input](#)
[Getting Input with Prompt](#)
[Getting Input with Input](#)
[Graph Screen Method](#)
[Reading Keypresses](#)
[Final Notes](#)

Getting Input

You can get input in three ways: Input, Prompt, and a graph-screen method. There are certain advantages and disadvantages to each command, and there are also certain situations where each command should be used. The first two are commands on the I/O menu of the prgm button (only while editing a program). The third has no single command. It has to be done manually. Below explains the third method a bit better.

Let us focus on the first two. Prompt and Input can be used with any variable, but some of the variables have to be entered in a certain way. If the variable is a string and you are using the Prompt command, the user must put quotes ("") around the value. However, both Prompt and Input require the user must also put curly braces ({}) around lists and square brackets ([]) around matrices.

Getting Input with Prompt

The Prompt command is the simplest way of getting user input. The Prompt command asks the user to enter a value for a variable, waiting until the user enters a value and then presses ENTER. When using Prompt, the variable that is being asked for will be displayed on the screen with an equal sign and question mark (=?) after it.

```
:Prompt variable
```

Because displaying what variable the value will be stored to does not tell the user what the variable will be used for, you can put a Disp command before the Prompt command to give the user some more insight into what an appropriate value for the variable would be. The Prompt command will be displayed one line lower, though, because the Disp command automatically creates a new line.

```
:Disp "Text"  
:Prompt X
```

Text
X=?

When you have a list of Prompt commands (and each one has its own variable), you can just use the first Prompt command and combine the rest of the other Prompt commands with it. You remove the Prompt commands and combine the arguments, separating each argument with a comma. The arguments can be composed of whatever combination of variables is desired.

The advantages of combining Prompt commands are that it makes scrolling through code faster, and it is more compact (i.e. smaller) and easier to write than using the individual Prompt commands. The primary disadvantage is that it is easier to accidentally erase a Prompt command with multiple arguments. So, instead of:

```
:Prompt A  
:Prompt Str1
```

Combine the Prompts and get:

```
:Prompt A,Str1
```

To use the Prompt command, you should first be in the Program editor for your program. In the Program editor, press the PRGM button, then arrow over to the I/O menu. Then, scroll down to Prompt and press ENTER. Now the Prompt command has been put into your program. You then have to type what variable(s) you want to prompt the user for (separating each one with a comma).

Final note: since the real-world applications take strings (like a username or a command) quite a bit, no-one uses Prompt because it forces the user to use quotation marks. A programmer would know to use it at the =?, but the casual user won't. 99.9% of the time, you see Input, which is discussed next.

Getting Input with Input

The other way to get input is to use the Input command. The Input command asks the user to enter a value for a variable (only one variable can be input at a time), waiting for the user to enter a value and press ENTER. The Input command, by default, does not display what variable the user is being asked for, but instead just displays a question mark.

```
:Input variable
```

Because just displaying a question mark on the screen does not really tell the user what to enter for input or what the input will be used for, the Input command has an optional text message that can be either text or a string variable that will be displayed alongside the input.

Only the first sixteen characters of the text message will be shown on the screen (because of the screen dimensions), so the text message should be kept as short as possible (a good goal is twelve characters or less). Don't worry about the user not having enough room, their input does word-wrapping.

```
:Input "Text",variable  
:Input Str#,variable
```

If the text message is longer than twelve characters or you want to give the user plenty of space to enter a value, you can put a Disp command before the Input command. You break the text message up and display it in parts. The Input command will be displayed one line lower, though, because the Disp command automatically creates a new line.

```
:Disp "Text"  
:Input "Text",variable
```

When you are just using the text message to tell the user what the variable being stored to is, the Prompt command makes it a byte easier. And, if there is a list of Input commands following the same pattern, you can reduce them to just one Prompt command.

```
:Input "A",A  
:Input "B",B
```

Replace with Prompt and get:

```
:Prompt A,B
```

The Input command can also be used another way. When you just put the Input command by itself, the graph screen will be shown and the user can move the cursor around. When the user presses ENTER, the (x,y) coordinates of the cursor will be stored to the X and Y variables, respectively.

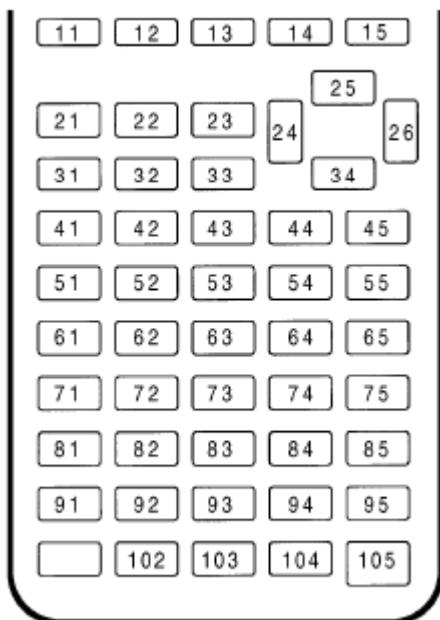
```
:Input
```

To use the Input command, you should first be in the Program editor for your program. In the Program editor, press the PRGM button, then arrow over to the I/O menu. Then, scroll down to Input and press ENTER. Now the Input command has been put into your program. You then have to type the text message and the variable that you want to ask the user for.

Graph Screen Method

This method has no automatic command. This must be done manually and, as you can imagine, takes a lot of space. If you value speed over design, then don't bother. Otherwise, [this guide](#) will show you how to do it.

Reading Keypresses



The getKey command is widely used in many programs because it allows the program to directly access user input. The getKey command returns the number of the last key pressed, and resets to 0 every time it is executed.

Every key has a number assigned to it, except for ON (which is used for breaking out of programs). The numbering system consists of two parts: the row and column. The rows go from one to ten, starting at the top; and the columns go from one to six, starting from the left. You just put the row and column together to find the key's number. The only confusing parts for beginners are the arrow key numbers.

When the getKey command is used, its value automatically resets to 0, so the next time it is used, it no longer returns the last pressed key. Because of this, the getKey command is usually stored to a variable. Storing getKey to a variable allows the program to keep track of which key was pressed,

taking different actions depending on what the key was. This opens up a variety of possibilities for the programmer. For example, using getKey inside a loop allows the program to wait for a keypress and store it:

```
:Repeat Ans  
:getKey  
:End  
:Ans→K
```

You can also put getKey in the condition of a loop, to make the loop repeat until any key or a particular key is pressed by the user. The same thing can be done with conditionals as well. This is useful if you don't want to store getKey to a variable, but you still want to have the user press a key. It's perfect for a "PRESS ANY KEY TO CONTINUE" screen.

```
:Repeat getKey  
:End
```

To use the getKey command, you should first be in the Program editor for your program. In the Program editor, press the PRGM button, then arrow over to the I/O menu. Then, scroll down to getKey and press ENTER. Now the getKey command has been put into your program.

Final Notes

Once you mastered this, you've mastered half of all programming. Literally. The other half is output. That's all a program is. A program gets user input and gets results, which are usually shown to you. Good luck to the rest of your TI-Basic venture

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/userinput>

Operators

Just like other programming languages, TI-Basic has the standard set of core operators built-in (math, relational, and logical), although they each have their own syntax and rules.

[fold](#)

Table of Contents

[Math Operators](#)
[Relational Operators](#)
[Logical Operators](#)
[Order of Operations](#)
[Test your knowledge](#)

Math Operators

There are five math operators: $+$, $-$, $*$, $/$, and $^\wedge$. Anybody who has ever done even basic math should know and recognize at least the first four operators, but for those who don't, their meaning is pretty straightforward:

- $+$ Adds two numbers together
- $-$ Subtracts one number from another
- $*$ Multiplies two numbers together
- $/$ Divides one number by another
- $^\wedge$

Raises a number to a power

There are two similar negative symbols on the TI-83 calculators — the subtraction symbol (the - key) and the negation symbol (the (-) key). These aren't interchangeable. However, it's almost always clear from an expression which one is being used, so the - symbol will be used to represent both throughout most of this guide.

Relational Operators

There are six relational operators: \equiv , \neq , \geq , \leq , \leq , and \leq . Just like with the math operators, these operators are used in almost every math class, and thus most people should know them.

- = $X=Y$ is true if X is equal to Y
- \neq $X \neq Y$ is true if X is not equal to Y
- > $X > Y$ is true if X is greater than Y
- \geq $X \geq Y$ is true if X is greater than or equal to Y
- < $X < Y$ is true if X is less than Y
- \leq $X \leq Y$ is true if X is less than or equal to Y

Here is a truth table of the various values:

X	Y	$X=Y$	$X \neq Y$	$X > Y$	$X \geq Y$	$X < Y$	$X \leq Y$
0	0	1	0	0	1	0	1
0	1	0	1	0	0	1	1
1	0	0	1	1	1	0	0
1	1	1	0	0	1	0	1

Because the calculator does not have a separate type for logical values (true and false), they are represented by the numbers 1 and 0. This becomes important when dealing with piecewise expressions.

Logical Operators

There are four logical operators: and, or, not(, and xor. Their interpretations are mostly intuitive when thinking about the meaning of the English word:

and

X and Y is true if both X and Y are true

or

X or Y is true if at least one of X and Y is true

xor

X xor Y is true if only one of X and Y is true

not(

$\text{not}(X)$ is true if X is false

Again, as with the relational operators, 1 is used to for 'true', and 0 is used for 'false'. It so happens that the logical operators treat all nonzero values as though they were 1 (true), so the expression '2 and 3' will be true just as '1 and 1'.

Here is a truth table of the various values:

A	B	A and B	A or B	A xor B	not(A)
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Order of Operations

The TI-83 series of calculators has nine priority levels for evaluating expressions. All the functions on a priority level will be calculated from left to right before moving on to the next priority level. Of course, calculations within parentheses are done first. Here is a table of the priority levels:

Priority Level	Functions
1	Functions that precede their argument (such as $\sqrt{}$ or $\sin()$, except for negation)
2	Functions that follow their argument (such as 2 or $!$)
3	\wedge and $\times\sqrt{}$
3.5	<u>Negation</u>
4	<u>nPr</u> and <u>nCr</u>
5	<u>Multiplication</u> , <u>division</u> , and implied multiplication
6	<u>Addition</u> and <u>subtraction</u>
7	The relational operators \equiv , \neq , \leq , \geq , \leqslant , \geqslant
8	The logic operator <u>and</u>
9	The logic operators <u>or</u> and <u>xor</u>
10	Conversions such as $\blacktriangleright \text{Frac}$

TI refers to the routine that determines order of operations as the Equation Operating System (EOS™). Unfortunately, this cool name hasn't become common usage.

Test your knowledge

Here are some sample problems on logical operators, in order of complexity. For the more difficult ones, it may be best to break them up into smaller parts and work in steps.

#	Question
1	0 and 1 or 1
2	0 and (1 or 1)
3	4 and -4 xor (.6 and 0)
4	not(1) xor (1 and 1 xor 1)
5	1 and 0 xor (6*4 and 0) or not(0 and 6)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/operators>

Calculator Linking

One of the most important features of the TI graphing calculators is their linking, where they communicate with another TI calculator or a computer across a link cable that is connected between them. There are a few different link cables that TI has created, and they each have their own advantages and disadvantages:

- **Graph Link** — This is the classic link cable that has been around since the TI-83 was first released. It works with every calculator, and it comes in black (for the PCs) or gray (for the Macs). It works with the Graph Link software, which doesn't work very well with the newer calculators.
- **USB Link** — This is the new link cable that is designed to be much faster, since it uses the USB port of a computer rather than the COMM port that Graph Link uses. Besides the port, it also only works with the TI Connect software instead of the Graph Link software.
- **Mini USB Link** — This is only available on the newer TI-84+/SE calculators, since it actually uses the second smaller USB port on the TI-84+/SE calculator instead of the usual I/O port. It works pretty much the same way the USB Link does, and in fact uses the same TI Connect software.

In addition to the official link cables, you can also make your own using parts of other cables. Putting together a link cable is a rather delicate operation, and requires a considerable amount of knowledge of electronics and linking. This isn't recommended unless you know what you are doing — if you screw up, you can really mess up your calculator!

Calculator to Calculator

There are three commands that you can use when linking one calculator to another: GetCalc(), Get(), and Send(). The GetCalc() command was designed such that you can receive a variable from another calculator; unfortunately there are very specific requirements for the sending calculator to actually send the variable (it must be in a power-saving state and cannot be executing an assembly program). Whilst this can seem a difficult task to actually create a fully functional and fun multiplayer game, the multiplayer page shows workarounds to make such a program achievable — the key to which is fully understanding the nature of GetCalc().

The Get(and Send(commands were created for use with the CBL (Calculator-Based Laboratory) and CBR (Calculator-Based Ranger) devices in math and science classes. These devices collect real-time data from various sensors that you can connect to them, and allow you to view and analyse the results. At the same time, they were originally used by the TI-82 calculator for receiving and sending variables respectively between calculators, and actually still operate in that capacity.

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/linking>

Controlling Program Flow

Controlling flow defines the order in which a program runs, what line of code will be executed next; to repeat or skip a group of commands. There are three main parts of controlling flow:

conditionals, loops, and branching. Each of these parts is used in different situations and to serve different functions. Together they are an integral part of all programs.

When dealing with conditionals and loops, the decision whether the conditional or loop will be executed is based on Boolean Logic — the principle that something can only be true or false at any given time. While any nonzero value is evaluated to true (i.e. it will be executed), a zero value is evaluated to false (i.e. it will be skipped over, not executed).

fold

Table of Contents

Conditionals

If Conditional

If-Then Conditional

If-Then-Else Conditional

Operators

IS>(and DS<(

Loops

For Loops

While Loops

Repeat Loops

Nesting Loops

Branching

Disadvantages of Branching

Reworking Branching to Remove Memory Leaks

So What Is Branching Good For?

Subprograms

Advantages & Disadvantages of Subprograms

Exiting Programs

Conditionals

Conditionals are used to make decisions in programs. The program can carry out different actions, depending on if certain conditions occur — directing the flow of program execution. Conditionals determine if code will be executed or not.

There are three different types of conditionals: If, If-Then, and If-Then-Else. There are certain advantages and disadvantages to each conditional, and there are also certain situations where each conditional should be used.

If Conditional

The first, and simplest, type of conditional is the If. It is used when you only want to execute one command. The If conditional needs the If command to work. The command that is immediately following the If conditional will be executed if the condition is true, but it won't be executed if the condition is false.

Format

:If condition

:Command

Because If conditionals are generally slow, you should replace them with Boolean conditionals

when you are just changing a variable. You take the condition that is in the conditional, put parentheses around it, and multiply it by the value that you are changing the variable (the value should be left off when it is one since it is unnecessary).

```
: If X=3  
: Y+2→Y  
Use Boolean Conditional  
: Y+2(X=3→Y)
```

The reason that this works is the condition will evaluate to one if it is true and zero if it is false. Since this value is then multiplied by the value that you are changing the variable, the changing value will stay the same if the value is one but it will become zero if the value is zero. So, the Boolean conditional is faster than an If conditional when the condition is true, but it will be slower when the condition is false because zero is still stored to the variable.

Boolean conditionals also have another advantage over If conditionals. When you have several Boolean conditionals that deal with the same variable, you can combine them into one Boolean conditional. Boolean conditionals can have multiple conditions that change the variable by different values. If you change the variable by the same value in two or more conditions, you can factor the value out by multiplication. This works best with large values.

```
: A+5(K=26→A  
: A-5(K=24→A  
Combine Conditionals  
: A+5((K=26)-(K=24→A
```

If-Then Conditional

The second type of conditional is the If-Then. It is used when you want to execute more than one command. Besides the If command, the If-Then conditional needs the Then and End commands to work. The Then command tells the calculator that there are multiple commands in the conditional to execute, while the End command signifies the end of the command block.

Format
: If condition
: Then
: Command(s)
: End

The commands immediately following the Then will be executed if the condition is true, but the commands won't be executed if the condition is false. Instead, program execution will continue after the End. Because If-then conditionals are twice as fast as If conditionals (they are larger, though, because of the added commands needed to use them), you might want to replace an If conditional with an If-Then conditional when speed is the top priority.

```
: If A=1  
: Disp "Hello  
Replace With If-Then Conditional  
: If A=1:Then  
: Disp "Hello  
: End
```

With the If-Then and If-Then-Else conditionals, you can put conditionals inside of each other (known as nesting). You can also put loops inside conditionals. When you have two or more If conditionals that have a common condition (i.e. a compound condition made using the logic operators), you should take the common condition out, make it into an If-Then conditional, and nest the If conditionals inside it.

```
:If A=1 and B=1
:C+2→C
:If A=1 and B=2
:D+1→D
Take Out Common Condition
:If A=1:Then
:C+2(B=1→C
:D+(B=2→D
:End
```

This will speed up the program execution when the If-Then conditional is false. Instead of testing each If conditional and its conditions, the If-Then conditional (and the nested If conditionals) will be skipped over if the first condition is false. Remember to put the closing End command for the If-Then conditional, otherwise you will get an error.

If-Then-Else Conditional

The third, and last, type of conditional is the If-Then-Else. It is used when you want to execute one or more commands if a condition is true and one or more other commands if the condition is false. This is equivalent to two separate If-Then conditionals with opposite conditions, but it is faster because there is only one condition test (since only one of the conditions can be true at one time). Besides the If command, the If-Then-Else conditional needs the Then, Else, and End commands to work.

Format
:If condition
:Then
:Command(s)
:Else
:Command(s)
:End

The commands between the Then and Else will be executed if the condition is true, while the commands between the Else and End will be executed if the condition is false. This is an important part of If-Then-Else conditionals because it determines what order you put the commands, whether they should go in the true or false part of the conditional.

When using an If-Then-Else conditional and only one command is executed if the condition is true or false, you can replace the If-Then-Else conditional with a simple If conditional. You switch the order of the commands so the false command comes first (because that command will be executed by default), and place the If conditional between the two commands. This primarily works when the commands are store commands, but it also can be used when you are building a string of text that you display.

```
:If B:Then
```

```

:"Hello→Str1
:Else
:"Goodbye→Str1
:End
Replace with If conditional
:"Goodbye→Str1
:If B
:"Hello→Str1

```

To put the conditional commands in your program, you need to first be in the Program editor. You press PRGM and then scroll over to EDIT. Once in your program, you press PRGM again and scroll over to CTL. The If, Then, and Else commands are in the first three spots (respectively), while the End command is in the seventh spot. You press ENTER to put the commands in your program.

Operators

Operators are used if you want to make compound conditions that are true depending on two or more conditions. When using operators, the left side is being compared to the right side. The operators can be used with any of the three different types of conditionals, as well as the Repeat and While loops.

There are two kinds of operators: conditional and logic. The six different conditional operators are: $=$, \neq , $>$, $<$, \geq , and \leq . The four different logic operators are: and, or, xor, and not. Conditional operators can be used (joined) with logic operators.

The $=$, \neq , $>$, $<$, \geq , and \leq operators all compare and test two conditions. $=$ returns true if the conditions are equal. \neq returns true if the conditions are not equal. $>$ returns true if the first condition is greater than the second condition. $<$ returns true if the first condition is less than the second condition. \geq returns true if the first condition is greater than or equal to the second condition. \leq returns true if the first condition is less than or equal to the second condition.

Format

```

:If condition = condition
:If condition ≠ condition
:If condition > condition
:If condition < condition
:If condition ≥ condition
:If condition ≤ condition

```

The one instance where you don't need the \neq conditional operator is when comparing a variable to zero. Because every nonzero value is treated as true, you don't need to compare if the variable's value is nonzero since any value will work. Instead, you can just put the variable by itself.

```

:If C≠0
Remove ≠ Operator
:If C

```

There is a simple truth table that is used to show how the logic operators work. The truth table is based on Boolean Logic, the principle that a condition can only be true or false. A true value is represented by one or any nonzero number. A false value is represented by zero. A and B are just conditions.

A	B	and	or	xor	not(A)
1	1	1	1	0	0
0	1	0	1	1	1
1	0	0	1	1	
0	0	0	0	0	

The and, or, and xor operators compare and test two conditions, while the not operator only tests one. and returns true if both conditions are true. or returns true if one or both conditions are true. xor returns true if either condition is true (but not both). not returns true if the condition is false.

Format

```
:If condition and condition
:If condition or condition
:If condition xor condition
:If not(condition)
```

One way that the not operator can be used is for switching something from true to false or on to off, and vice versa. When dealing with a variable, not inverts the variable's value; so you should use not instead of comparing a variable to zero because not returns true when the variable is zero. At the same time, don't try to use not in every condition because there are many ways of writing a condition.

```
:If A=0
Use not Operator
:If not(A)
```

The not operator is also used when applying DeMorgan's Law. DeMorgan's Law can be used for conditions in which there is an individual not operator around two separate unary conditions (i.e. they don't have conditional operators) joined by the and or or operators. It allows you to remove the second not operator and then change the and to or, and vice versa.

```
:If not(A) and not(B)
Use DeMorgan's Law
:If not(A or B)
```

The and and or operators can be replaced using math logic. Since and is only true when all the conditions are true, you can multiply the conditions together for the same effect (you can leave off the multiplication sign). Only one condition has to be true for or to be true, so adding the conditions together works as well. For conditions that have operators attached to them, you just put parentheses around them so they are treated as Boolean values. However, math logic is somewhat slower compared to the logic operators.

```
:If A and B
:If A or not(B)
Replace Operators
:If AB
:If A+not(B)
```

When using the and operator, if the first condition is false, the second condition will not be tested. The and and not operators have the highest importance (precedence) of the logical operators, so they are evaluated first. This is useful when you have a condition that combines the and and or operators (where the and operator comes first), because you don't need to include parentheses around the and operator. However, parentheses are sometimes needed simply to provide clarity.

```
:If (A=1 and B=2) or (A=2 and B=1)
Remove Parentheses
:If A=1 and B=2 or A=2 and B=1
```

To put the operators in your program, you need to first be in the Program editor. You press PRGM and then scroll over to EDIT. Once in your program, you press 2nd and MATH. The conditional operators are in the TEST menu, while the logic operators are in the LOGIC menu. You press ENTER to put the commands in your program.

IS>(and DS<(

Two specialized conditional commands are available: IS>() and DS<(). These commands are equivalent to If conditionals, except the next command will be skipped when the condition is true. They have the variable update built-in, so they are smaller than using regular If conditionals.

The IS>() and DS<() commands each take two arguments, but they differ in functionality. The first argument is the variable, and it can be a real variable (A-Z or θ). The second argument is the value, and it can be either a number, variable, or expression.

IS>() adds one to the variable (increments it by one), and compares it to the value. The next command will be skipped if the variable is greater than the value, while the next command will be executed if the variable is less than or equal to the value.

```
Format
:IS>(variable,value)
:Command
```

DS<() subtracts one from the variable (decrements it by one), and compares it to the value. The next command will be skipped if the variable is less than the value, while the next command will be executed if the variable is greater than or equal to the value.

```
Format
:DS<(variable,value)
:Command
```

These commands are not without problems, however. Because the skipping feature is usually not needed, you will have to make sure that the value is always greater than (or less than) the variable, so that the next command is executed. This is not always possible to do. An undefined error will occur if the variable doesn't exist before the command is used, which happens when the DelVar command is used. Finally, these are not looping commands, so they shouldn't be used in that manner.

To put the IS>() and DS<() commands in your program, you need to first be in the Program editor. You press PRGM and then scroll over to EDIT. Once in your program, you press PRGM

again and scroll down the CTL menu until you find the commands. You press ENTER to put the commands in your program.

Loops

Loops cause a segment of code to repeat until a stated condition is met. Instead of having to write out something or do an action several times, you just do it once and put it inside a loop.

There are three different kinds of loops: For, While, and Repeat. There are certain advantages and disadvantages to each loop, and there are also certain situations where each loop should be used. For loops should be used when you know how many times the loop will be executed, whereas Repeat and While loops are the converse. The For loop is the fastest of the three loops.

For Loops

The For loop takes four arguments: the variable (A-Z or theta), the starting value, the ending value, and the increment. It counts from the starting value to the ending value at the specified increment.

The variable is used to keep track of how many times the For loop has been executed. Because it is set to the starting value when the For loop begins, you don't need to initialize the variable before. The ending value is the value that the variable ends at. The increment determines how much the variable's value will be increased each time through the loop. The default increment is 1, so the increment can be left off when it is 1 (it is optional). The increment can be positive or negative.

After each time the For loop is executed, the variable is checked to see if it is equal to or greater than the ending value. If the variable is, then the loop is exited and program execution continues after the End command. (The End command determines the boundaries of the loop.) If the variable isn't, the variable is incremented by the increment and the loop is executed again.

Format

```
: For(variable,start,end[,increment])  
: Command(s)  
: End
```

One of the common uses of For loops is making delays. Although you can use the Pause command, this brings the program to a halt and the user has to press ENTER to get out of it. With a For loop, you can make a small delay that will only last as long as you want it to last and it doesn't require the user to do anything. You just use an empty For loop (no commands inside of it). The larger the difference between the starting and ending values, the bigger the delay.

```
: For(X,1,200)  
: End
```

Sometimes you might want to prematurely exit out of a For loop (stop it before it is completely finished). You can do this by changing the variable inside the loop. You just need to make the variable larger than the ending value.

```
: For(A,5,100)
```

```
:110→A  
:End
```

To put the For loop command in your program, you need to first be in the Program editor. You press PRGM and then scroll over to EDIT. Once in your program, you press PRGM again and scroll over to CTL. You then scroll down to For (or press the 4 key) and press ENTER. The End command can be found in the same menu, just lower at the seventh spot on the menu (press the 7 key).

While Loops

A While loop executes a block of commands between the While and End commands while the specified condition is true. The condition is tested at the beginning of the loop (when the While command is encountered), so the loop will be skipped entirely if the condition is false when the loop is first entered. To ensure that the loop will be executed, you need to declare the values of the variables in the condition before the loop.

After each time the While loop is executed, the condition is checked to see if it is false. If it is false, then the loop is exited and program execution continues after the End command. If the condition is true, the loop is executed again.

```
Format  
:While condition  
:Command(s)  
:End
```

When using While loops, you have to provide the code to break out of the loop (it isn't built into the loop). If there is no code that ends the loop, then you will have an infinite loop. An infinite loop just keeps executing, until you have to manually exit the loop (by pressing the ON key). In the case that you actually want an infinite loop, you can just use 1 as the condition. Because 1 is always true (based on Boolean Logic), the loop will never end.

```
Format  
:While 1  
:Command(s)  
:End
```

To put the While loop command in your program, you need to first be in the Program editor. You press PRGM and then scroll over to EDIT. Once in your program, you press PRGM again and scroll over to CTL. You then scroll down to While (or press the 5 key) and press ENTER. The End command can be found in the same menu, just lower at the seventh spot on the menu (press the 7 key).

Repeat Loops

A Repeat loop executes a block of commands between the Repeat and End commands until the specified condition is true. The condition is tested at the end of the loop (when the End command is encountered), so the loop will always be executed at least once. This means that you sometimes don't have to declare or initialize the variables in the condition before the loop.

After each time the Repeat loop is executed, the condition is checked to see if it is true. If it is true, then the loop is exited and program execution continues after the End command. If the

condition is false, the loop is executed again.

Format

```
:Repeat condition  
:Command(s)  
:End
```

When using Repeat loops, you have to provide the code to break out of the loop (it isn't built into the loop). If there is no code that ends the loop, then you will have an infinite loop. An infinite loop just keeps executing, until you have to manually exit the loop (by pressing the ON key). In the case that you actually want an infinite loop, you can just use 0 as the condition. Because 0 is always false (based on Boolean Logic), the loop will never end.

Format

```
:Repeat 0  
:Command(s)  
:End
```

To put the Repeat loop command in your program, you need to first be in the Program editor. You press PRGM and then scroll over to EDIT. Once in your program, you press PRGM again and scroll over to CTL. You then scroll down to Repeat (or press the 6 key) and press ENTER. The End command can be found in the same menu, just lower at the seventh spot on the menu (press the 7 key).

Nesting Loops

One important aspect of loops is putting them inside other loops (known as nesting). Besides nesting any of the different kinds of loops inside each other, you can also nest loops inside conditionals. When nesting loops, you need to remember to put the appropriate number of End commands to close the loops.

The easiest way to keep track of lots of nested loops is to code the first part, add an End immediately after the conditional, and then hit [2ND][DEL] on the line with the End, then hit [ENTER] a lot of times.

Branching

Branching allows the calculator to jump from one point in a program to another. Sometimes you don't want every part of the program to be executed. You may want to skip over a certain part of a program if a certain condition occurs.

Branching uses the Lbl and Goto commands. Lbl and Goto work in pairs; you need to have both for branching to work. The Lbl command specifies a location in a program. The label can be any one or two alphanumeric character combination (from A-Z, 0-9, and θ), but ideally you want it to be only character to save memory. The Goto command causes program execution to jump to the specified label with the same character combination, and then continue from there.

Format

```
:Lbl character1[character2]  
:Goto character1[character2]
```

When using branching, you have to provide the break-out code (it isn't built-in). If there is no code that ends the branching, then program execution will continue indefinitely, until you manually exit it (by pressing the ON key). If conditionals are commonly used, but in the case you want infinite branching, you should instead use a While or Repeat loop.

```
:Lbl A  
:Goto A  
Replace with Loop  
:Repeat 0  
:End
```

To put the Lbl and Goto commands in your program, you need to first be in the Program editor. You press PRGM and then scroll over to EDIT. Once in your program, you press PRGM again and scroll over to CTL. You then scroll down to Lbl (or press the 9 key) and press ENTER. The Goto command can be found in the same menu, just lower at the tenth spot (press the 0 key).

Disadvantages of Branching

Although branching may seem like a good alternative to loops, it should be used sparingly. Branching should only be used when a loop isn't practical and when something only happens once or twice. This is because branching has several disadvantages associated with it.

The biggest disadvantage of branching is that it's slow. When the calculator reaches a Goto command, it stores the label name in memory and goes to the beginning of the program. It then searches through the program until it finds the Lbl command with the matching label name. If the label is deep within the program and you have a large program, this can bring the program to a crawl.

Another disadvantage of branching is that it can lead to memory leaks when used to exit conditionals or loops (anything that uses an End command). The calculator stores the End command in memory, and it is only released when the calculator reaches it.

If the conditional or loop is exited with branching, however, the End command is never released from memory, and the calculator will continue using that memory. If this is done enough times, the calculator will eventually run out of memory, causing a memory leak. When there's less memory, the program also runs more slowly. Memory leaks don't have any real affect on the calculator, as the memory is released when the program exits.

The last disadvantage of branching is that it makes program code difficult to read and maintain. While loops are straightforward, following a set pattern, branching can lead to anywhere in a program. Trying to figure out how branching affects the program code can cause some serious headaches.

Reworking Branching to Remove Memory Leaks

One of the simplest memory leaks that occurs is using branching to exit out of a loop when a certain condition of an If conditional is true. If the loop is an infinite loop (i.e. Repeat 0 or While 1), you should take the condition from the If conditional and place it as the condition of the loop. This allows you to remove the branching, since it is now unnecessary.

```
:Repeat 0  
:getKey→B  
:If B:Goto A  
:End:Lbl A
```

```
Place Condition in Loop
:Repeat B
:getKey→B
:End
```

Of course, the only reason that this memory leak fix is possible is because of the If conditional (since the If conditional doesn't need a closing End command). When dealing with an If-Then or If-Then-Else conditional, you will have to rework the conditionals so the branching has its own If conditional. Depending on how many commands there are in the conditionals, you might be able to just use an If conditional or you might need to use an If-Then conditional.

```
:If B:Then
:Disp "Hello
:Goto A
:End
Use Separate If Conditionals
:If B
:Disp "Hello
:If B
:Goto A
```

This memory leak fix will work most of the time, but it isn't applicable when one of the values of the variables in the condition is changed by one of the commands inside the condition. The way to get around this is by using another variable for the If conditional that the branching uses. You initialize the variable to zero, assign the variable whatever value you want in the conditional, and then check to see if the variable is equal to that value in the branching conditional.

```
:If A=1:Then
:3→A:4→B
:Goto A
:End
Use Another Variable
:Delvar C
If A=1:Then
:3→A:4→B:π→C
:End
:If C=π
:Goto A
```

So What Is Branching Good For?

Despite its many disadvantages, Lbl and Goto statements actually have their uses. For example, you may want to have a label at the end of the program that you Goto everywhere you want to exit the program. This is useful if you have a lot of clean-up (such as deleting large temporary variables) every time the program exits.

```
If K=45:Goto Q
...
Lbl Q
DelVar [A]DelVar L1
ClrHome
```

Goto statements are also good in programs that call themselves very many times. Every time a Repeat or While statement is encountered, the program has to set aside a portion of memory to remember about that statement. In recursive programs, this can add up (a good example is a recursive program to fill in an arbitrary shape). Gotos require no such overhead, and if the program is small, they're not as slow as they are in larger programs.

Just remember that since Goto-Lbl constructs are slow when the label is far from the beginning of the program, and you shouldn't use them in speed-critical situations. Also, they make your program hard to read for when you or anyone else edits it, especially if they jump backwards.

Subprograms

Subprograms are programs called from inside other programs (at any time while the program is running). Although they are listed in the program menu and can be executed independently like any other program, subprograms are primarily designed to do a particular task for the other program.

The prgm command is used to execute another program as a subprogram. You insert the prgm command into the program where you want the subprogram to run, and then type (with the alpha-lock on) the program name. You can also go to the program menu to choose a program, pressing ENTER to paste the program name into your program.

```
Format  
:prgmname
```

To create a subprogram, you take the code from the parent program and put it in a new program. When naming your subprograms, you should try to name them Zparentn or θparentn, where parent is the name of the parent program and n is the number (if you have more than one). Because subprograms are relatively unimportant by themselves, you want them to appear at the bottom of the program menu.

When the subprogram name is encountered during a program, the program will be put on hold and program execution will transfer to the subprogram. Once the subprogram is finished, program execution will go back to the program, continuing right after the subprogram name.

Although subprograms can call themselves or other subprograms, this should be done sparingly because it can cause memory leaks if done too much or if the subprogram doesn't return to the parent program. Branching is local to each program, so you can't use Goto in one program to jump to a Lbl in another program. All variables are global, so changing a variable in one program affects the variable everywhere else.

To put the prgm command in your program, you need to first be in the Program editor. You press PRGM and then scroll over to EDIT. Once in your program, you press PRGM again and scroll over to CTL. You then scroll down until you get to the prgm command, and press ENTER to put the command in your program.

Advantages & Disadvantages of Subprograms

There are several advantages of using subprograms. First, and foremost, subprograms reduce program size by eliminating redundant code. Instead of having to type the code multiple times for a task that occurs more than once in a program, you just type it once and put it in a subprogram. You then call the subprogram whenever you want to perform the task in your program.

Second, subprograms increase program speed by making programs as compact as possible.

You separate conditional tasks from the program (they either happen every time or they are skipped over), and put them in a subprogram; you then call the subprogram instead. This improves program speed because the calculator doesn't have to go through all of the conditional code anymore.

Third, subprograms make editing, debugging, and optimizing easier. Instead of going through the entire program, looking for the code you want to change, you can focus on one subprogram at a time. This makes the code more manageable, allowing you to more thoroughly look at each subprogram and to better keep track of which subprograms you have looked at.

Lastly, subprograms are reusable, allowing multiple programs to share and use the same code. Breaking a program into smaller, individual subprograms, which each do a basic function or task, allows other programs to use those subprograms. Consequently, this reduces program size.

The primary disadvantage of subprograms is that there are additional programs that the user needs to use your program. If you give someone your program, you will have to also give them your subprograms. Your program won't work properly anymore if somebody deletes your subprograms or forgets to include them with your program. Although this is mostly out of your hands, users will think your program is at fault.

The simple solution to this problem is to put the subprogram back in your program when you finish it. This should only be done if the subprogram was just used once or twice and it won't slow the program down. All you have to do is paste the code from the subprogram in place of the program call. You could also put all of the programs used into a group and distribute it as so.

Exiting Programs

Exiting programs (terminating the execution of a program) is done with the Return and Stop commands. There are certain advantages and disadvantages to each command, and there are also certain situations where each command should be used.

When the Return and Stop commands are used in a program, they both stop the program execution and return you to the homescreen. If they are encountered within loops, the loops will be stopped. Return and Stop function differently, however, when used in subprograms.

The Return command will stop the subprogram, and program execution will go back to the calling program, continuing right after the subprogram call. The Stop command will stop the program execution of the subprogram, as well as the calling program, and return you to the homescreen; the program will stop completely. So, Return should generally be used instead of Stop.

```
:ClrHome  
:Input "Guess:",A  
:Stop  
Replace Stop with Return  
:ClrHome  
:Input "Guess:",A  
:Return
```

You don't have to put a Return command at the end of a program or subprogram if you can organize the program so that it just naturally quits. When the calculator reaches the end of a program, it will automatically stop executing as if it had encountered a Return command (the Return is implied).

```
:ClrHome
:Input "Guess:",A
:Return
Remove the Return
:ClrHome
:Input "Guess:",A
```

To put the Return and Stop commands in your program, you need to first be in the Program editor. You press PRGM and then scroll over to EDIT. Once in your program, you press PRGM again and scroll over to CTL. You then scroll down until you get to Return or Stop, and press ENTER to put the command in your program.

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/controlflow>

User Settings



The image shows the Mode screen of a TI-83 calculator. It lists several options with their current settings:

- Normal, Sci, Eng, Float 0123456789, Radian, Degree, Func, Par, Pol, Seq, Connected, Dot, Sequential, Simul, Real, a+bi, re^θi, Full, Horiz, G-T
- RectGC, PolarGC, CoordOn, CoordOff, TblStart=0, GridOff, GridOn, ΔTbl=1, AxesOn, AxesOff, IndPnt: Ask, LabelOff, LabelOnDepend: Auto, ExprOff, ExprOn, Ask
- TABLE SETUP, Tbl1Start=0, ΔTbl=1, Ask

The TI-83 series of calculators has many options to select from that influence either the effect of commands or the way numbers or graphs are displayed. Outside a program, these can be changed by accessing the Mode, Format, or TblSet screens (shown above), and selecting the correct options. When editing a program, going to these screens and choosing an option will instead paste a command that sets that option for you.

These commands are given below, divided by the screen where it is found:

Mode Settings (MODE)

- Normal, Sci, and Eng determine how large numbers are displayed.
- Float and Fix determine how decimals are displayed.
- Radian and Degree determine which form of angle measurement is used.
- Func, Param, Polar, and Seq determine the graphing mode.
- Connected and Dot determine the default graphing style.
- Sequential and Simul determine how multiple equations are graphed.
- Real, a+bi, and re^θi determine how complex numbers are displayed (and affects ERR:NONREAL ANS)
- Full, Horiz, and G-T determine how and if the screen is split.

Graph Format Settings (2nd FORMAT)

- RectGC and PolarGC determine how coordinates of the cursor are displayed and stored.
- CoordOn and CoordOff determine whether the coordinates of the cursor are displayed at all.
- GridOn and GridOff determine whether the grid is displayed.
- AxesOn and AxesOff determine whether the X and Y axes are displayed.
- LabelOn and LabelOff determine whether the X and Y axes are labeled (if they are)

- ExprOn and ExprOff determine whether the equation graphed or traced is displayed.
- Time, Web, uvAxes, uwAxes, and ywAxes (visible in Seq mode) determine the way sequences are graphed, the default being Time.

Table Settings (2nd TBLSET)

- IndpntAuto and IndpntAsk determine whether values of the independent variable in the table are calculated automatically.
- DependAuto and DependAsk determine whether the values in the table are calculated automatically for all equations.

Miscellaneous Settings (2nd CATALOG)

- DiagnosticOn and DiagnosticOff determine whether the statistics r and/or r^2 are displayed by regressions.
- FnOn and FnOff determine whether equations are graphed.
- PlotsOn and PlotsOff determine whether plots are graphed.
- Pmt_Bgn and Pmt_End determine whether payments are done at the beginning or end of a period with the finance solver.

Using these settings in a program

A fair amount of these settings are important to programmers because, if set to the wrong value, they can easily mess up the program. At the beginning of the program, therefore, it's a good idea to set these settings to the correct value. At the very minimum, programs that use the graph screen should set AxesOff if necessary, since AxesOn is the default and a very common setting. This is a part of program setup.

However, another important consideration is that it's somewhat rude to change the user's settings without permission, so your program should change as little as possible. How to reconcile these diametrically opposite goals? There are several ways that work for different settings:

Use GDBs (Graph DataBases)

The graph screen settings can be backed up in (and retrieved from) a GDB file by the StoreGDB and RecallGDB commands. If you store to a GDB at the beginning of the program, and recall from it at the end, you will have preserved all settings that deal with the graph screen.

Change math settings implicitly

Instead of changing settings like the Degree/Radian or the Real/a+bi setting, you can use certain commands that will force calculations to be done with that setting regardless of the mode. For example, you can use the degree symbol or radian symbol to make ambiguous calculations like $\sin(30)$ into unambiguous ones like $\sin(30^\circ)$. Similarly, by adding $0i$ to a number, you force it to be complex, so that calculations done with it will never cause an ERR:NONREAL ANS (even if you're in Real mode).

Ignore uncommon settings

You might ignore settings that are too uncommon to matter. For example, setting the Full command is unnecessary, because very few people would ever use a split screen, and people that do probably will also figure out why your program breaks when they do so.

Be rude when you must

For something like Float, there's no way to avoid changing the user's settings in a way you can't restore. If you have to change a setting so your program works, do it, and mention the issue in the readme. If you changed a setting to an uncommon value, change it back to "Float" (in general, to the default value) when you're done.

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/settings>

Memory Management

The TI-83 and TI-84 series of [calculators](#) feature a considerable amount of memory available for storing [variables](#), [programs](#), [groups](#), and even [assembly applications](#) (the last two are only available on the TI-83+/TI-84+/SE calculators). However, as you start putting those things on the calculator, the memory slowly gets used up, and if you don't do anything, the calculator will eventually run out of memory — giving you the dreaded [ERR:MEMORY](#) error.

Before getting into memory management, it is important that you know there is a major difference between the TI-83 calculator and the rest of the TI-83 and TI-84 series of calculators. In particular, the TI-83 just has RAM (Random Access Memory) memory, while the rest of the calculators have RAM memory as well as Flash ROM (Read-Only Memory) memory, also known as archive memory.

RAM is the faster memory of the two, and it is what is primarily used whenever you run a program. Its one downfall is that it tends to get cleared very easily after crashes. Most people have probably noticed this when their calculator crashed, and they turned it back on, and it said RAM cleared. In addition, variables in RAM can be overwritten accidentally by a program that uses them.

Flash ROM, on the other hand, is the more reliable memory, and it is what is used when you want to store a program for long-term. Its one downfall is that you can't access something stored in ROM. The only exception to this is assembly programs and applications, which are programmed in the calculator's own programming language and thus can access anything on the calculator.

There are several different commands you can use for managing your calculator's memory:

- [**DelVar**](#) — DelVar is useful for deleting variables, which is the most obvious way to manage memory, and it is what most people are interested in. The DelVar command can delete any variable that you want, with exception to specific elements of a matrix or string or [system variables](#).
- [**ClrList/ClrAllLists**](#) — Similar to DelVar, ClrList and ClrAllLists only work with [lists](#) and they set the dimensions of one list or all lists to zero respectively. This essentially causes the list(s) to be deleted, since you can't do anything with a zero element list.
- [**Clear Entries**](#) — When executing programs or doing math on the calculator's home screen, the calculator keeps a history of these entries (you can cycle through them by pressing 2nd ENTER). If you do a lot of stuff on the home screen, the entries history can become rather large.
- [**Archive/UnArchive**](#) — When using variables and programs, you need to move them from archive to RAM; and when you are done using them, you move them back to archive. While you can archive programs on the home screen, that is not possible from inside a program (although you can use an [assembly library](#) to do that).
- [**GarbageCollect**](#) — As you archive and unarchive variables and programs, the calculator keeps storing things until it eventually needs to clean the archive memory. Rather than simply leaving this until the calculator finally forces you to do it, which takes a fair amount

of time, you can run the GarbageCollect command periodically.

Since the TI-83 calculator only has RAM memory, it does not have the Archive, UnArchive, and GarbageCollect commands. If you plan on porting a program to the TI-83, you shouldn't use any of these commands, since they will cause the program to be corrupted. In the case of lists, however you can use the SetUpEditor command instead of UnArchive to get around this problem.

Also note that trying to use DelVar or ClrList with an archived variable does not work, and will actually return an ERR:ARCHIVED error.

Accessing the Memory Menu

When accessing a variable or program from the memory menu, you press 2nd MEM. You then select 2:Mem Mgmt/Del and press one to display a scrollable list of all the files on the calculator. You use the up and down keys to move the cursor on the left. On the top lines of the screen you will see how much free RAM and ARC (archive) memory there is.

Once you have found a file you want to delete, press DEL. If the file is not a variable, the calculator will prompt you to confirm the deletion, and you have to select 2:Yes. Once you have found a file you want to archive, press ENTER. An asterisk will appear to the left of the file name, indicating that it is archived. Some files, such as applications, will not allow you to unarchive them since they can only reside in ROM.

Archiving may sometimes not be possible, however, if the calculator does not have sufficient free ROM available. This occurs primarily when a person can't bring themselves to delete a file because they feel like every file is important. At this point, the only option is to delete some files off of their calculator to make room. As part of memory management, a good policy is to keep the calculator's memory organized and to delete any files that you don't need.

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/memory-management>

Time And Date Commands

The TI-84 Plus and TI-84 Plus SE, which have a built-in hardware clock, introduce several commands for dealing with time. Some of these rely on the built-in clock, while others are used for formatting times and dates and could technically have been introduced on earlier calculators. However, the only time/date command that is available on the pre-84 calculators is dbd().

All of these commands except dbd(can be found only through the command catalog (2nd CATALOG). dbd() can also be found in the Finance menu — 2nd FINANCE on the TI-83, and APPS 1:Finance... on the TI-83+ or higher.

Despite its name, the Time command has nothing to do with the clock. It is a mode setting for sequence graphs.

Low-Level Commands

- **startTmr** — This command returns the current value of a timer that is updated every second when the clock is enabled. This value doesn't correspond to any actual time, but

Date Commands

- **setDate()** — Sets the current date (year, month, and day). If the clock is enabled, this date will be updated as needed.
- **getDate** — Returns the current date

can be used with `checkTmr` to get a time difference.

- **checkTmr** — `checkTmr(X)` is equivalent to `startTmr-X`. This can be used to get the time elapsed since `startTmr` was used.
- **ClockOn, ClockOff** — Enables or disables the hardware clock.
- **isClockOn** — Tests if the clock is enabled or not.

Time Commands

- **setTime** — Sets the current time, in hours, minutes, and seconds. If the clock is enabled, this time will be updated every second.
- **getTime** — Returns the current time as the list {hours, minutes, seconds}. This command is unaffected by time format.
- **setTmFmt** — Sets the time format - 12 hour, or 24 hour.
- **getTmFmt** — Returns this time format setting.
- **getTmStr** — Returns the current time as a string, affected by time format (though you can override it with an optional argument).

as the list {year, month, day}. This command is unaffected by date format.

- **setDtFmt** — Sets the date format - 1 for month/day/year, 2 for day/month/year, or 3 for year/month/day.
- **getDtFmt** — Returns this date format setting.
- **getDtStr** — Returns the current date as a string, affected by date format (though you can override it with an optional argument).

Time/Date Manipulation

- **timeCnv** — Converts a number of seconds into a list of {days, hours, minutes, seconds} representing the same time lapse.
- **dayOfWk** — Returns the day of week (Sunday through Saturday encoded as 1 through 7) of a specified date.
- **dbd** — Returns the number of days between two dates — this command is available on all calculators, not just the 84+/SE.

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/time-and-date>

Planning Programs

Before writing any of the code for a program, you should carefully plan out the program. This may seem like an unnecessary step, time that could be better spent, but it will pay major dividends in the end. Planning not only results in better quality programs, but many times it will also cut down the coding time (since you don't have to waste time rewriting the program) — a win-win situation!

The first thing you want to do when planning a program is to decide what the program will do. Beginner programmers often say that they want to create a cool game, but they don't get much farther than that. For them to have a real chance of creating their program, they need to determine what the objective of the program will be, and then build off of that. For program ideas, see the [projects](#) page.

When coming up with an idea for a program, you should try to be realistic about the [limitations](#) of TI-Basic, and what a program can and can not do. For example, a game that needs lots of speed to be worthwhile for the user to play, such as Phoenix or Mario, really isn't very practical in TI-Basic beyond only moving a few things on the screen at any one time. In addition to speed, TI-Basic also suffers from limited graphics capabilities.

Once you have determined what the program will do, you need to decide what features the

program will have. This can include: potential program options, the interface ([home screen](#) or [graph screen](#)), main menu, an about screen, user help, and any other things you may want. The more thorough you are with planning your program, the easier the coding will be; it is to your benefit to do a good job.

If you can't come up with any ideas for your program or you are unsure of if the ideas that you have come up with make sense, you should get input from the TI community. The three most friendly, active user forums are:

- [United-TI](#)
- [MaxCoderz](#)
- [Cemetech](#)

Since these are the people that are going to be primarily using your program when it is finished, you want to ask them to evaluate your program ideas and to offer some constructive criticism. They might also be able to give you some new ideas that you never thought of.

Even if you thoroughly plan a program and get community input, it's simply not possible to think of everything up front. While making changes later on when a program is in heavy development can be a lot more work than making those changes at the beginning, there's nothing wrong with changing or modifying your plans, if you believe the program will be better with the change(s).

Research Before Coding

Before doing any coding, you should do some research to determine what the best algorithms are for use in your program. One of the most common problems is a subpar algorithm, where the algorithm was not fully thought out or it doesn't work together with the other parts of the program the way it should.

When you do research you ensure that the algorithm is appropriate and that it will work effectively. This helps eliminate flaws in your algorithm, which can cause a multitude of errors if left undone. If you think your program through before you actually do any of the coding, you can save yourself lots of time because you don't have to do several rounds of testing and debugging to get your program to work the way it should.

One of the ways to test an algorithm and how effective it will be in your program is to take a very small problem and trace by hand how your chosen algorithm would work in that situation. This allows you to see if the algorithm will actually work in the given situation.

If the algorithm doesn't work, you can immediately start looking for another algorithm. This saves you lots of potential time because you would have to come up with another algorithm had you just started coding it. Only when you are confident with the algorithm should you start the coding.

Translate It Into Pseudocode

The next step in the process is turning the program plans into pseudocode. Psuedocode involves using English (or whatever language you speak) in place of the TI-Basic code to describe what the program will do to perform the desired functions and tasks. This prevents you from getting caught up in the TI-Basic syntax, allowing you to more clearly focus on the program.

You should first start by looking at the big picture of the program and then break it down into smaller and smaller details. Using an outline as the base, this means you would put the most important things first and then gradually add everything else. This allows you to mentally picture what the program is going to look like and to make sure you don't forget anything.

An important part of creating useful pseudocode is adding comments throughout. It is very easy to get lost in your logic or have problems come up that you don't have any idea on how to resolve. Besides telling you what the code is supposed to do (i.e. making coding easier), it will also force you to slow down and think through the logic of your program. Still, comments are only as good as you make them.

Use Many Small Programs While Coding

A single large program quickly becomes unwieldy and difficult to manage. While you're still editing the program, it's best to keep it in many small pieces. When you're done, you can combine them into one program again.

One of the benefits of this approach is that you can convert pseudocode into a main program almost right away. For example, imagine this pseudocode program:

```
Main Menu - user enters difficulty, etc.  
Initialize variables  
Main Loop:  
    Player movement  
    Draw player  
    Enemy movement  
    Draw enemy  
    Check Win/Loss Condition  
End Main Loop  
If we won the game  
    Display win message  
Otherwise  
    Display loss message  
Cleanup
```

You could translate this into a basic program almost directly. Here's how we do it (note that we don't write any code yet):

```
prgmMAINMENU // user enters difficulty, etc.  
prgmINITVARS // initialize variables  
Repeat Z  
    prgmMOVEPLR // moves player  
    prgmDRAWPLR // draws player  
    prgmMOVEENMY // moves enemy  
    prgmDRAWNMY // draws enemy  
    prgmWINLOSE // sets Z to 1 or 2 if we won or lost  
End  
If Z=1:Then // we won  
    prgmWEWON // says "You win!"  
Else  
    prgmWELOST // says "You lose!"  
End  
prgmCLEANUP // delete used variables
```

As you progress in writing the actual code, you create and edit each individual program (for example, you would create and edit prgmMAINMENU and write a menu in that program). Of course, if these sub-programs are big enough, you can split them up into their own sub-sub-programs in the same way.

When all the subprograms are finished, the program will work as it is, in 50 or so pieces (so you can test for bugs and tweak the individual programs). However, if you want to release your program, you probably don't want there to be 50 small programs to send. You can use the Recall feature (press [2nd][STO] to get to it) to combine the programs.

Go through the main program. Every time you get to a sub-program call, clear that line and press [2nd][STO]. The Recall option will come up. Press the [PRGM] key and select the appropriate sub-program from the EXEC menu. The calculator will paste that sub-program into the main program. When you're done, all the code is in your main program (and you can delete the now-unnecessary sub-programs)!

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/plan>

Commenting Your Code

When you are in the process of writing a program, everything about the program (i.e., variables, program structure, branching labels, etc.) is fresh in your mind and it's easy to remember. But what would happen if you stopped working on the program for a year, and then decided to start working on it again?

All or most of your previous knowledge of the program would be gone, and you would have to spend some time (or lots of time, depending on the program) to figure out how the program works again. While this is only a hypothetical example, this problem is a real concern for lots of people who have multiple projects going at any one time.

Thankfully, this problem can be easily prevented by simply adding comments to your code. A comment is a brief note that describes some functionality or feature of your code. You don't need to comment everything, but the two main things that you should comment are:

- **Program Structure** — Loops and branching with appropriate label names
- **Variables** — What their purpose is and any possible values

The amount of free RAM is the only limit on comment size, but you should generally try to keep your comments clear and concise. In addition, if your comment is more than a couple lines in length, you should split it up into multiple lines. This makes it easier to read and update, if needed.

Text Comments

There are a couple different ways that you can add text comments to your code, with each having their own advantages and disadvantages. The first way is to make the comment a string literal (i.e., place a quote before the comment text), and then just place it on its own line.

```
: "Comment here"
```

The advantage of this comment method is that it is extremely simple to use and update. You can make your comment almost anything you want (the two characters you can't use are the store command and a quote), and the calculator just reads it as a string.

The disadvantage of this method is that it prevents you from using the Ans variable, since the comment will be stored to Ans when the calculator reads it. The comment also slows the program down because the calculator has to execute it each time.

The second way to add a text comment to your code is by placing the comment in a conditional or loop, and using zero as the condition. Based on Boolean logic, the condition will always be false, which causes the calculator to not execute the conditional or loop, and subsequently skip right over the comment nested inside of it.

```
:While 0  
:Comment here  
:End
```

The advantage of this comment method is that it doesn't mess with any of the variables, and you can use the store command and quote character. The disadvantage is that it takes up some additional memory to use the conditional or loop, and this problem only worsens the more comments you use.

Indenting Code

Another way to comment code is by indenting it, which allows you to easily identify control structures and blocks of code. Just like how there is a built-in colon that denotes the beginning of each new line, you can place your own colons before any statements on a line.

```
:While 1  
::Disp "Hello  
::Disp "Goodbye  
:End
```

Although there is no restriction on how many colons you can place at the beginning of a line, one colon is generally sufficient. However, if adding two or three colons helps you better visualize the code, then that's what you should go with.

Descriptive Variables

Yet another way to comment code is by using descriptive variables that reflect where and how they are used. This is primarily related to using the real variables (A-Z and θ), since they are the most commonly used variables, and are much smaller and faster than other variables.

Of the real variables, the standard variables and situations are:

- I and J for the looping variable in For(loops
- X and Y for the X and Y screen coordinates respectively.
- K for storing the keypress with the getKey command.

Each of these variables is mnemonic — for example, K is the first letter in keypress — making them fairly easy to remember.

Advanced Uses

You can modify the text comments so that you can turn them on or off. You just use a conditional with a variable as the condition, and then change its value from false (i.e., the comments are off) to true (i.e., the comments are on). Based on Boolean logic, the easiest system for the variable value is one for true and zero for false.

You also need to display the comments on the screen, so that you can read them during

program execution. If you are using the [home screen](#), you should use the [Pause](#) command and its optional argument. While program execution is halted until the user presses ENTER, the Pause command allows you to display the entire comment on one line, and you can even scroll the comment left or right to read it, if necessary.

```
:If A:Pause "Comment here
```

You should use the same variable for all of the comments, so that the comments work in unison. The variable can be whatever you want, but the simplest variable to use is a real variable (A-Z and). You just need to remember to set the variable to zero at the beginning of the program, so that the comments are turned off by default.

See Also

- [Planning](#)
- [Debugging](#)
- [Code Conventions](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/comments>

Code Conventions

Code conventions are a set of guidelines for writing programs, primarily focusing on the structure and appearance of the code. Although code conventions are generally subjective and informal, just the individual preferences of each programmer developed over time, there are several common conventions that most programmers follow and are considered to constitute good programming practice.

Following code conventions not only makes your code consistent, helping to make it easier to read and understand, but also eliminates a lot of the difficulty in maintaining it. Code conventions are also important in group projects, where multiple people are working together and everybody needs to be on the same page.

Naming

The general convention for naming programs, [subprograms](#), [labels](#), and [custom lists](#) is to choose a name that tells you what it is (in the case of programs and subprograms) or that relates to how it is used (in the case of labels and lists).

For example, if your program is a Minesweeper clone, then a good program name is something like MINES. If your MINES program has a subprogram, start it with θ or Z so that it appears at the bottom of the program list.

If you have a [highscore](#) function built-in to your game, then you should use a custom list that is related to your program's name (i.e., LMINE would make sense for the MINES game mentioned above).

In the case of a label name, you should make it mnemonic — Lbl UP would be an appropriate choice for the code that moves the screen up.

Formatting

The general convention for formatting code is to place related statements together on the same line. This is most applicable with variable declarations, If-Then conditionals, and short loops, although you can certainly put whatever statements that you want.

The way you get multiple statements on the same line is by separating each one with a colon, which is also used to denote the beginning of each new line. The program editor on the calculator allows sixteen characters per line, so if the statements are wider than that, they will cause the line to wrap around to the next line (and however many more lines are needed).

Structure

The general convention for structuring code is to use a loop, except for those situations where using a loop is impractical; in those cases, using a Goto is the preferred convention.

When deciding which loop to use, you need to look at what its functionality will be. If you want the loop to be executed a set number of times, then you should use a For loop. If you want the code inside of the loop to be executed at least once, then you should use a Repeat loop.

The While loop is very similar to the Repeat loop, so the way to decide when to use either one is by thinking of the loop condition. If the loop is going to keep running as long as the condition is true, then you should use a While loop. If the loop is going to run until the condition is true, then you should use a Repeat loop.

Variables

The general convention for using variables is to use the most appropriate variable whenever possible. There are several different kinds of variables available, including reals, lists, matrices, and strings, and they each have their own time and place.

Reals are used for keeping track of one value, and because they are both small and fast, you should use them before using other variables. Lists are used for keeping track of multiple values, and because they can be created, you should use them for saving highscores and other important data.

Matrices are used for keeping track of two-dimensional values, which means you should use them for making maps on the screen. Strings are used for keeping track of characters, which means you should use them when you want to display text on the screen.

Sample Program

The following sample program is a modified form of the program on the movement page. The main things you should notice are the real variables grouped together on the first line and the use of the Repeat loops for the code structure.

```
PROGRAM:MOVEMENT
:4→A:8→B
:Repeat K=105
:Output(A,Ans,"X
:Repeat Ans:getKey→K:End
:Output(A,B," "
:A+(Ans=34)-(Ans=25
:Ans+8(not(Ans)-(Ans=9→A
:B+(K=26)-(K=24
:Ans+16(not(Ans)-(Ans=17→B
```

: End

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/code-conventions>

Debugging Programs

Going through the process of debugging your programs may seem like a lot of unnecessary work, but it is better that you find and correct the errors than having the user of your program tell you that your program doesn't work. On this page, we've tried to simplify debugging for you by breaking it down into general steps that you can follow. Just remember to always strive for bug-free programs.

fold

Table of Contents

- [Backup & Document](#)
- [Simplify the Code](#)
- [Analyze The Error](#)
- [Debugging Tools](#)
- [Be Prepared For Setbacks](#)
- [Get Outside Help](#)

Backup & Document

Before debugging, make a backup of your program. This is to ensure that you don't lose your program in its current form. During the debugging process it is easy to mess up your program, overwriting or deleting necessary pieces of code. If you just spent several hours working on your program, you don't want to have all of that work wasted.

You should also backup your program before changing anything. This is so you will have a version of your program that you can return to if you mess up your program while doing further debugging. Instead of having to undo any coding mistakes that you made, you can just go back to the last updated version of your program.

You also want to document your program and the variables you use. This is important because it makes it easier to come back to the program in the future (if you need to). Documenting a program entails listing what the program does, how it does it, and anything else that might be important. For documenting the variables, you should list out all of the variables, including what they are used for and what values they can have.

Simplify the Code

You want to remove any code that is not related to the problem. If you are unsure of what code is related to the problem, look at what its function is. If the code is changing variables or controlling program flow, then it may be important to keep it. If it is just print statements or checking conditionals, you can probably remove it without it affecting the code.

Once you have identified the code that contains the error, you will want to create a new program and put the code in it. If the error still exists, you will want to remove parts of the code, repeatedly testing the code afterwards each time. If the error disappears, you know that the last

part of the code that you removed is what caused the error.

One of the most common problems that programmers make when debugging their code is making lots of changes at once. This is usually not very effective because it can lead to other problems within the code or, if it results in removing some of the errors, you don't know which changes corrected the errors. It also tends to make the debugging process very unorganized because you will have to re-test parts of the code, and sometimes you won't even know which parts need to be re-tested. What you should do instead is fix each problem individually and then test the program. This will take you longer, but your program will have fewer errors and problems.

This can further be improved upon by breaking the program up into modules, testing and debugging the individual modules separately from the main program. The advantage of testing the modules by themselves is that it is easier to find errors. Instead of having to look at the whole program, you can look at an isolated part. Once you have removed all of the errors in the module, you don't have to test it again. After going through all of the modules, you then just have to debug the main program that calls the modules.

Analyze The Error

When you are trying to fix an error in your program, you will want to gather as much data about it as possible. To do this you should run your program several times, keeping track of when the error occurs, where it is happening, and what type of error it is.

Once you think you have a solid grasp of the error, you will want to repeatedly test the code that contains the error. If you can consistently produce the error, you will know that you are on the right track to fixing it. If you can't produce the error, however, that means you need to do more extensive analysis.

One of the most effective ways of debugging programs is to walk through the code. This involves sitting down with a printed version of the program, carefully going through its logic. Walking through the code will allow you to see potential solutions to the errors in your program.

Once you have a potential solution, you will want to look at how it should work in the code, and then test it to see if it actually works like you believe. The more thorough you are at analyzing the solution, the easier it will be to tell if it will fix the error. Once you know that the solution will fix the problem, you can then apply it to your program.

Looking at the variables and their values is essential when you are debugging. You will want to take note of how they gradually change throughout the program. If you know which part of the program contains the error, you can then check the variables that are used within that part. Make sure the variables contain values within their correct limits. If they don't, you need to go to the code before and after the error, and check the value of the variables. Make sure the variables are functioning properly, and that you aren't doing anything to change them.

When writing code you will often make assumptions about it. You need to be aware of these assumptions when you are debugging, so you can make sure they are sound. If your program is not functioning properly, you should test all of your assumptions. One of the most common assumptions that programmers make is that the variables they are using are working properly. Most programmers will thoroughly debug their code, but they're not as thorough when debugging variables. Debugging can become very frustrating if you don't thoroughly debug both.

Debugging Tools

When you first start debugging your programs, you will want to check to see that the errors you are getting aren't because of a misuse of a command or a misunderstanding of a command's

arguments. You should consult the TI-83+ manual to see what the syntax of the command is and how to use it properly. Many errors that you receive can easily be remedied if you just consult the manual. In fact, if you use the manual when you are programming your program, you can avoid a lot of the typos and superficial errors during debugging.

The TI-Basic language has a rather useful feature for debugging programs: when it comes across an error while running your program, it will give you an error menu — telling you what the error is and giving you the option to see where the error is in your code. You will want to take the information that it gives you, and then see if you can figure out why it's producing an error.

After the error occurs, you should recall some of your variable's values to see what they are. This might give you an indication of what the error is (if a variable is not in the range it's supposed to be), where in a For(loop you are (just recall the variable you used for the loop), and many other helpful hints.

Many times you will make a simple mistake, such as forgetting to close a string or leaving an argument off of a command. You have to be careful, though, because sometimes the error that the calculator reports isn't the actual error in your code. You might have errors in your code that the calculator doesn't notice because it is still valid code, and it isn't until later that the errors cause problems. These errors are very hard to track down.

Using print statements is one of the oldest and most tested methods for debugging programs. If you come upon a problem in your program, and you can't seem to figure out what's causing it, you should add several print statements throughout the code. When you run your program, you will be able to see what is actually happening. You can do this to see the code that the program is running or to see what values the variables have.

If you don't like using print statements or you think they aren't effective enough, an alternative that you can try is inserting breakpoints (pause statements). Once you think you have identified the code that has the error, you can set breakpoints around it. After restarting the program, look at what happens to the code before and after the breakpoints. If this doesn't give you enough information about the error, you might need to use more breakpoints. The advantage of using breakpoints is that you can pause the execution of your program, allowing you to slowly look at the program and at what's causing the error. Two of the best places to put breakpoints are in program flow statements and inside loops.

Changing the code to solve a problem is an effective way to debug programs. You go to where the problem is, and you start changing parts of the code. You can change whatever you want to, but you should have a general idea of how the code works. If you don't know your code very well, this can cause even more errors. You should also remember to make a backup of your code before you change anything.

If changing the code doesn't help in solving the problem, you might want to try creating test code. Creating test code allows you to focus on the problem, making it easier to see and fix. You look at the code in your program that has the problem, and then you create a new program with similar code. You then experiment with that code, switching things around or adding code to it, looking at how it affects the operation of the code. You can also do this as a way to learn some of the more confusing aspects of the TI-Basic language.

Be Prepared For Setbacks

After working on debugging a program for a prolonged period of time, with no progress and no new ideas on how to fix the error, you stop being able to effectively debug your program.

One of the simplest remedies is to just take a break from debugging and do something else. Take a walk outside around the neighborhood or take a nap. Many times after taking a nap, you will suddenly get the answer to the problem in an epiphany. In addition, it is just a good rule to

take frequent breaks from programming so that you don't get burned out so easily.

Although it may seem like debugging your program will be a monumental task requiring lots of work, it is essential that you do it. If you are to release your program to the public, you don't want users complaining that it doesn't work correctly or that it contains errors.

Another cause of feeling overwhelmed is if you are not very good at debugging. When you are just starting out, you will be able to fix the simple or obvious errors, but you will have a hard time tracking down some of the more complicated errors. The only way to get around this is by repeatedly debugging programs until you have figured out the errors for yourself. The more debugging you do, the better you become at it. It just takes practice.

Often when you are debugging a program, and it seems like you just can't find the error in it, you will stop thinking logically and start thinking irrationally. Your only desire is to get the program to work correctly, so you decide that you will do that by whatever means necessary. If you still can't figure out the problem, you might start blaming the calculator or the TI-Basic language.

While blaming the calculator or the TI-Basic language will provide you with temporary relief concerning the error in your program, you have to remember that they don't do any of the thinking. They just follow what your program says to do. You are the one that is responsible for the code that you produce.

Get Outside Help

If you have tried everything that you know to do and you are still unable to fix the problem, you should now start looking for outside help. You should ask other programmers or go to programming forums. Either one of these should be able to help you with your problem.

Asking other programmers for help is a good alternative to getting mad at yourself because you can't figure out the problem. Because you wrote the code, you may make assumptions or have biases when debugging it. You know the code so well that you can't be objective. When another programmer looks at it, though, they don't have any of those hang-ups. In addition, when you're explaining the problem to the other programmer, many times the solution will come to you. Asking the other programmer for help also benefits the other programmer because they improve their confidence debugging programs.

If you asked another programmer for help and they could not find and fix the problem, you should then go to programming forums. The advantage of programming forums is that several programmers are working together, building off of each other's ideas. This is the ideal situation because the more people looking at the code, the greater the chance that the problem will be found and fixed.

Here are some programming forums that you should go to if you ever need help with a problem:

- [United-TI](#)
- [Maxcoderz](#)
- [Calcgames](#)
- [Cemetech](#)
- [Omnimaga](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/debug>

Setting Up A Program

fold

Table of Contents

General Settings

Numeric Notation

Decimal

Real/Complex Mode

Screen Display

Home Screen

Graph Screen

Initialize Variables

Putting It All Together

At the beginning of a program, you typically setup everything that the program will use while it's running. Of course, there are lots of things that you may decide to include in your individual program setup, but the three main things that you should include are: the home screen, graph screen, and initializing variables. There are also some general, but crucial mode settings that should be taken care of.

General Settings

There are some general mode setting that you'll want to pay attention to. Most of those should be what you want, but there is always a chance that a program forgot to switch back to the standards, or that the user was playing around with the calculator.

In the mode menu, there are probably four different modes you need to worry about. These are numeric notation, decimal, real/complex mode, and screen display.

Numeric Notation

How numbers are displayed/returned: Normal, Scientific, and Engineering. Scientific will have one digit on the integer side, and Engineering will have two digits. The standard is Normal.

Decimal

In programs that use pinpoint precision numbers or require complex formulas or calculations, the number of decimals returned can greatly affect the program. Float will automatically adjust to the number of digits the calculator considers significant. Fix 1-9 will fix the calculator to display 1-9 digits, no matter what. This means that the calculator may sometimes give weird results such as 3.100000, or pi=3.

Real/Complex Mode

The default mode, Real, will give ERR:NONREAL ANS whenever a complex number is obtained as a result. If you want to use complex numbers, you should change this setting to a+bi or re^θi (the distinction between these two is only a display one).

If you're going to be using complex numbers, you should switch away from Real mode. Otherwise, it's an inessential setting. Switching to Real mode doesn't have any real (he he) purpose to it, since it doesn't provide any extra functionality - unless of course you like it when your calculator throws errors.

Screen Display

This affects the screen display. Full is probably the one of the only ones you have ever seen. Horiz displays a horizontal split-screen, with the graph on top and home screen on bottom. G-T displays a vertical split-screen, with graph on left and table on right. The standard is Full.

Home Screen

Since the home screen that your program uses is the same home screen that the rest of the calculator uses, the previous program call(s) and any other text is typically still displayed on the screen. Obviously, you don't want to be displaying text and have it interrupted by other text, so you need to clear the home screen. The ClrHome command is what you use.

When using the ClrHome command, you simply place it on a line. The whole home screen will be cleared of any text; there's no way to clear a smaller portion of the ClrHome because it takes no arguments.

```
:ClrHome
```

Graph Screen

The typical TI calculator user uses the graph screen to graph, which means they use axes, stat plots, Y= equations, and sometimes the grid. They might also like drawing things with the drawing commands or the Pen. However, while working in a game in use of the graph screen, you really do not want these functions to appear, which would completely mess up your program.

First, you need to disable all these annoying thing by the following code:

```
:ClrDraw      // Clears the graph screen of all its contents
:AxesOff     // Disables X and Y axis scaling view
:FnOff       // Disable Y= equations
:PlotsOff    // Disables stat plots from appearing
:GridOff     // Disables grid from appearing
```

After that, you setup the window dimensions to use a friendly window. This not only makes drawing much easier, but it is faster and smaller. One way to do this is shown below:

```
:0→Xmin:1→ΔX
:0→Ymin:1→ΔY
```

Initialize Variables

If you have any important variables that you use in the main program loop, you should initialize them here, so the program will be able to use them and not have a delay. This is especially important with large variables (such as lists, matrices, and strings), since initializing those variables inside the main program loop will definitely have an impact on its speed.

```
:{1,2,3,4→A
:[[1,2][3,4→[A]
:"1234→Str1
```

Putting It All Together

Putting all the parts of program setup together, here is a typical way to start a program:

PROGRAM: SETUP

```
:ClrHome  
:ClrDraw  
:AxesOff  
:FnOff  
:PlotsOff  
:GridOff  
:0→Xmin:1→ΔX  
:0→Ymin:1→ΔY  
:{1,2,3,4→A  
:[ [1,2][3,4→[A]  
:"1234→Str1
```

Of course, you only have to include the things that you actually use. If you don't have any important variables to initialize, then simply leave that off. In the same fashion, you don't have to clear the clear the home screen if your program just runs on the graph screen.

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/setup>

Cleaning Up After a Program

fold

Table of Contents

- [Deleting Large Variables](#)
- [Restoring the Graph Screen](#)
- [Clearing the Home Screen](#)
- [List Editor Cleanup](#)
- [Putting It All Together](#)

Imagine you just finished playing a round of Blackjack, and now you're back on the main screen. You enjoyed the game, but something is just not right. Not only is there text on the home screen, but there's graphics on the graph screen, and it appears that there's some leftover variables taking up a considerable amount of your precious memory. It seems that the Blackjack game forgot to clean up after itself.

Cleaning up after a program is one of the most important parts of any game. A quality game features good gameplay, but more importantly it doesn't leave the calculator in disarray afterwards so the next program that is run isn't affected by it. While program cleanup can involve whatever the game programmer wants, there are a few standard parts to it.

Deleting Large Variables

The first, and arguably most important, part of program cleanup is deleting variables. After a

program finishes running, it should delete any large variables that it created during its execution. The program should only keep variables if they are used for storing important information, such as highscores or map data. You can delete a variable using the DelVar command (provided that the variable is not in the archive).

The user does not want to have their memory cluttered up with lots of variables because it makes scrolling through the memory menu that much harder. They also don't want to lose any of their memory because it prevents them from using it for any other things they want to do (such as running other programs).

Restoring the Graph Screen

After deleting large variables, the next part of program cleanup is to restore the graph screen. Besides clearing the graph screen (using the ClrDraw command), you should recall the graph database (GDB) variable that has the previous window and graph format settings stored in it. (Please note that GDBs do not contain text, graphics, or stat plots.)

You want to make sure to clear the graph screen when exiting programs because this ensures that the next program that the user runs won't have to deal with whatever text or graphics your program left behind. It also helps the user because they won't have to manually clear the graph screen themselves.

At the beginning of a game that uses the graph screen, select whichever GDB you want to use (GDB0 through GDB9) and then use the StoreGDB command to save the window and graph settings into that GDB. Now when the program is finished executing, recall that GDB with the RecallGDB command to recreate the graph screen with the previous graph and window settings that were stored in it. You should then delete the GDB.

Clearing the Home Screen

Once the graph screen is restored, the next part of program cleanup is to clear the home screen using the ClrHome command. Clearing the home screen ensures that the next program the user runs will not have to deal with whatever text the program left behind. It also helps the user, because they will not have to manually clear the home screen by pressing the CLEAR key; you have already done it for them.

Besides clearing the home screen when cleaning up, you should also remember to remove the "Done" message that shows up after a program finishes executing. This "Done" message is a clear indicator that your program just finished running (which can be bad if you are in class and your teacher is near by), and it also does not look very good.

When you display text, a number, a variable, or an expression with a display command (either Disp or Output() on the last line of the program, you can remove the command and just put argument by itself. The argument will be displayed instead of the "Done" message that is normally displayed after a program finishes executing, and it will also be stored into the Ans variable.

```
:ClrHome  
:Disp "Hello  
Remove Disp  
:ClrHome :"Hello
```

If you do not display any text on the last line, or you do not have any particular text that you want to be shown, you can still remove the "Done" message by just putting a single quotation mark. This will have the same effect, but there will be no text and the cursor will be placed on

the second line.

```
:ClrHome  
Put a quote  
:ClrHome :"
```

In addition to removing the "Done" message, this text also acts as a way to clear the Ans variable. For example, if you had a large variable stored in Ans (such as [A]), which subsequently would cause Ans to also be large, this text would make Ans release that excess memory back to the calculator. You could also add the Clear Entries command before the final text just for good measure.

To remove the "Done" message without moving the cursor (slightly larger):

```
:ClrHome  
:Output(1,1, "           //no space, just a quote
```

List Editor Cleanup

If you used the SetUpEditor command for lists that your program uses, that also causes the list to appear in the "List Editor" the next time the user accesses it (STAT>Edit...). This probably isn't desired behavior, because it looks unprofessional, and because your program's list is likely to contain highscores and other data of that nature.

To fix this, add a SetUpEditor without any arguments to the end of your program. This will reset the List Editor to the default settings (it will show the contents of L1, L2, ..., L6). This isn't perfect, since the user may have been editing his own lists there, but it's the best you can do, since TI-Basic can't find out about the user's previous settings.

Putting It All Together

Putting all the parts of program cleanup together, here is a typical way to end a program:

```
PROGRAM:CLEANUP  
:SetUpEditor  
:DelVar Str1DelVar [A]  
:RecallGDB 1  
:DelVar GDB1  
:ClrDraw  
:ClrHome :"
```

Of course, you only have to include the things that you actually use. If you don't use any large variables, you don't have to delete them. In the same fashion, you don't have to clear the graph screen and restore the graph screen settings if your program just runs on the home screen.

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/cleanup>

Usability (User-Friendliness)

Imagine you are using a program for the first time. You have no prior knowledge about the program; someone just put the program on your calculator without giving you any instructions and now you are trying to figure out how to use it. After literally pressing all of the keys on the calculator and trying all sorts of key combinations, you give up and delete the program.

This example isn't based off any one particular program, but it does resonate with lots of program users who have had a similar experience. What this problem really is about is poor user-friendliness — more commonly known as usability. The definition of usability is simply how easy it is for people to use a program.

While usability can take on many different forms, there are some essential things that you can do to make a program more user-friendly.

In-Program Help

Probably the easiest way to make a program user-friendly is by including some in-program help. While you ideally want your program to be so easy to use that a user can simply pick it up and figure out how to play it, not every game is so straightforward, and the average user probably needs some help.

The best place to include help in a program is as one of the options in the program's main menu. When the user comes across the menu, they will see the help option and they can select it to view the help. The help does not need to cover every minute detail about the program, but rather just explain the objective of the game and detail what keys are used for controls.

```
:Menu("Some Game", "Option 1", 1, "Option 2", 2, "Help", 3  
...  
:Lbl 3  
:Disp "Game Objective  
:Disp "Key = Function
```

Because most people do not like using help unless they have to, you should try to limit your help to one or two screens at most. At the same time, if you have an extremely complex game with all sorts of features and lots of keys are needed to operate it, then it would be appropriate to include help for all of those things. The general guideline is that the amount of help needed correlates to the size of the game.

Protect the User

The next thing you can do to make a user-friendly program is to protect the user from themselves. Often times in a program you will want to think about what could go wrong and try to either prevent it from happening or tell the user what's wrong. Preventing it from happening involves you, the programmer, programming in safety protections for the user so that they aren't even aware that something went wrong.

Say the program calls for the user to type in a number between 1-1000, and the user types in 5000. If your program just goes on with this value, it will probably crash at some point later on. Rather, it's necessary to check the value, and display an error message and ask for the number again if it's wrong. The error message does not need to be complicated or long — just enough so that you can provide some direction on what input you are expecting the user to enter.

```
:Disp "Enter a Number  
:Input "Between 1-1000", A
```

```
:While A<1 or A>1000
:Disp "Must Be 1-1000!
:Input "Number",A
:End
```

Of course, just checking to see that the number is in the appropriate range is sometimes not enough. You might also want to check to see whether the user tried to enter text or a list for input. Because there is no viable way to perform those checks when dealing with a real variable, a better option would be storing the input to a string and performing the validation on it, and then converting the string to a real variable.

Include Helpful Features

Another part of making a user-friendly program is to include helpful features. Since the target audience is often in high school, a feature sure to be appreciated is a "teacher key." This is a special key that the user can use to quickly exit the program. When the teacher comes around, they then want to be able to get back to the home screen so that they don't get their calculator taken away.

This problem is quite easy to prevent with a teacher key. In every program there is a main loop that runs throughout the life of the program. You need to add a check for whatever teacher key you want at the place in the main loop where you check for user input. While you can have any key function as the teacher key, the community standard is usually MODE or DEL. (It is probably best for you to continue this so that users don't have to deal with figuring out which key is the teacher key.)

```
:While main loop not finished
:Display something
:Perform calculations
:Get user input
:If teacher key pressed, exit program
:End user input
:End main loop
```

Progress Indicators

In games that use maps, the program has to go through the list of maps and then load the appropriate one for the user to use. Depending on the size and number of maps, this can take a while. If the user doesn't know what is going on, they probably will think the program stalled or something else went wrong.

While there are a couple different ways you can cut down on the loading times for maps (see subprograms and compression), the easiest way to solve the problem is by simply telling the user what is going on and showing the user some progress. You don't have to do anything fancy (in fact, you probably shouldn't because that would just waste valuable memory), just something to help the user understand the situation.

For example, say you are randomly placing mines throughout the map (it's a Minesweeper game), you then could just display a "Placing Mines" message on the screen and have a loop for the progress indicator that matches the current map loading:

```
:Output(3,2,"Placing Mines
:For(X,1,20
```

```
: // fill the map with mines
:Output(4,6,5X
:End
```

Follow the KISS Principle

The last important point of usability is following the KISS principle. For those who haven't heard of KISS, it is an acronym which stands for Keep It Simple Stupid. The basic point of KISS is to not clutter your program with unnecessary features and useless fluff. It also entails making the program easy to figure out for those who don't have access to a readme.

It is not uncommon to see a TI-Basic math program (i.e., quadratic solver) that has a menu, about screen with scrolling credits, and includes some game in case you somehow get bored solving quadratic equations. While those things by themselves aren't bad, they are completely inappropriate in a math program. There is a certain elegance that comes with "programs that do one thing and do it well." This is known as the Unix philosophy, and should really be what every program strives for.

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/usability>

Portability

Portability is the ability to run a program on more than one calculator, with little to no changes made to the program — you can literally transfer the program to a calculator, and then start using it. This ability is important because all of the TI-83 series calculators are very similar in TI-Basic support and calculator hardware, so people expect when they find a TI-83 series program it will work on their calculator.

There are five primary things that you need to consider when making a program portable:

- Making sure not to use assembly programs
- Making sure not to use new commands
- Making sure not to use undocumented functionality
- Making sure not to use extra characters
- The calculator's memory and speed

Assembly Programs

Although assembly programs allow you to make your programs look nice, and to use functionality that isn't normally possible or viable in TI-Basic (such as creating parallax scrolling using xLIB), they are not portable because they need to be compiled to work on each calculator.

This is because assembly programs are programmed in the calculator's own machine language, and use memory addresses that are specific to a particular calculator model. This means, for instance, that a TI-83 assembly library that inverts the graph screen will not work on any of the new TI-83 series calculators.

The TI-83 also uses a different format to run assembly programs than the other TI-83 series calculators: `Send(9prgmNAME)`. This use of the Send(command does not work on any of the other calculators, and in fact will result in a ERR:SYNTAX error. Instead, the rest of the TI-83 series calculators provide three commands — Asm(, AsmPrgm, and AsmComp(— for running

and compiling shell-independent assembly programs.

Two additional commands for running assembly programs have been added to the TI-84+ and TI-84+SE calculators: OpenLib(and ExecLib. These can be used for running routines from Flash application libraries that have been specifically written for use with them; the only application so far is usb8x, which is used for interfacing with the USB port.

Apart from use of these last two commands, however, most assembly programs ought to be compatible between the TI-83+/SE and TI-84+/SE

New Commands

With each release of a TI-83 series calculator, TI has added new commands to the calculator. The TI-83+ calculator introduced Archive, UnArchive, and GarbageCollect, which are designed to work with the Flash memory available on the calculator. This is in addition to the assembly commands that were mentioned earlier.

The TI-84+ and TI-84+SE calculators introduced several new time and date commands, some of which use the new built-in clock, while others are used for formatting times and dates; and the aforementioned OpenLib(and ExecLib for running routines from Flash application libraries. The new OS (2.30 or later) also includes some additional commands for statistics: Manual-Fit, invT(, LinRegTInt, and x²GOF-Test(.

Undocumented Functionality

Along with documented changes, different calculators and OS versions have some undocumented differences. These are given below grouped by the first calculators they occur on:

TI-83+ or higher:

- **Large font on the graph screen** — Use the syntax Text(-1, row, column, text) to display text in the large font instead of the typical small font associated with the graph screen.
- **Fast circle drawing** — If you put a complex list, such as {i}, as the fourth argument of Circle(, the circle is displayed using its symmetries to only do 1/8 of the trigonometric calculations; this cuts the display time down to only about 30%.

OS version 1.15 or higher:

- **The % Command** — The % symbol is an undocumented command that is a useful shortcut for percents — it divides by 100, so it will convert numbers to percentages. For example, 50% will become 50/100 or 1/2, which is just what 50% should be.
- **The sub(Command** — If only one argument is given, and it contains an expression that evaluates to a real or complex number or list of numbers, the argument will be divided by 100. A simpler version of the % command above.

TI-84+ and TI-84+ SE:

- Using the Text(command for small text will sometimes erase the row of pixels below the text (usually not noticeable, when text is displayed on an already-white background). See the command itself for more information.

Extra Characters

At three points in TI-83 series history, TI allowed more characters to be used in TI-Basic. However, this means that if you use a new character, it will not work on older calculator models.

- **First group:** This includes the lowercase letters, Greek letters, and international characters. These characters will work with all calculators starting with the TI-83+, but there may be some issues with computer programs such as the TI Program Editor.
- **Second group:** The \sim $@$ $\#$ $\$$ $\&$ $\`$; \backslash $|$ $_$ $\%$ characters were introduced only with OS version 1.15 (and will work on all higher versions).
- **Third group:** The \dots \angle β x^y T \leftarrow \rightarrow \uparrow \downarrow x \int $\sqrt{}$ \blacktriangleleft \blacktriangleright \blacksquare characters and subscripts $_0$ through $_{10}$ were introduced only with OS version 1.16 (and will work on all higher versions).

Calculator Memory & Speed

The TI-83 is the oldest TI-83 series calculator, and it only has 27KB of RAM and a 6MHz processor. A program cannot really even take up the whole 27KB of RAM, since there is the in-game use while running the program. In addition, the 6MHz processor is slower than any of the other calculator processors, so if a game is only marginally playable on the TI-83+SE (with its much speedier 15MHz processor), then there is no way it would even be playable on the TI-83.

This primarily affects large, complex games like the RPG's made by Kevin Ouellet, but can also affect games that need a lot of speed to be fun. For example, if you have a Mario-like game where you need to keep track of and display multiple enemies on the screen, this can be quite time-consuming on the TI-83. In fact, the game would probably slow to a crawl, and you would spend most of your time waiting for things to load.

This problem doesn't only plague the TI-83, but also the TI-83+. Because the TI-83+ only has 184KB of memory (24KB RAM and 160KB Flash), each of the aforementioned RPG's by Kevin Ouellet would literally take up all of the available memory on the calculator: Metroid II, for instance, takes up over 123KB in Flash, and you need to have several of the almost fifty programs unarchived in order to actually play the game.

The TI-83+ also only has an 8MHz processor (which is just marginally faster than the TI-83's 6MHz processor), while the TI-83+SE and TI-84+SE each have a 15MHz processor. So, if a game is specifically tailored to run on those two calculators (meaning that the speed of the game is just fast enough), there is no viable way that the TI-83+ would be able to run the game at a sufficient speed (even taking into account optimization).

Thoughts to Consider

There are some additional ways that you can avoid portability problems:

- Use SetUpEditor instead of UnArchive for a list — this is better, and doesn't lose compatibility with the TI-83.
- If possible, replace all lowercase letters from your program with lowercase stat variables from the VARS>Statistics... menu, or just use uppercase letters everywhere.
- Instead of using dayOfWk(), use the day of week routine which uses the dbd() command instead.
- Place all of the calculator specific code into subprograms that the main program calls: one program is your game functions that work on the respective calculators and the other program is the primary all-calculator code for the program.

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/portability>

Optimization

A dictionary would define *optimization* as the process of making something better. In the field of

TI calculator programming, it refers to improving code to use less memory, whether as program size or in the size of variables used, or to run faster. It should be your goal, in virtually all cases, to make your programs as optimized as possible.

Line-by-Line Optimization

Optimization techniques fall naturally into two classes. The first, which we'll call "line-by-line optimization", refers to ways of rewriting a line of code, or several lines, so that it does basically the same thing, but is smaller or faster. Typically, each such optimization doesn't have a huge effect. But since many lines can be improved this way, these optimizations add up over the entire program to produce a smaller and faster result.

Read the [basic techniques](#) of line-by-line optimization. Then, consult the following pages to see techniques for specific topics:

- [Displaying Text](#)
- [Storing Variables](#)
- [Deleting Variables](#)
- [User Input](#)
- [Exiting Programs](#)
- [Logic Operations](#)
- [Conditionals](#)
- [Loops & Branching](#)
- [Math Operations](#)
- [Using the Ans variable](#)
- [The Graph Screen](#)

Alternatively, you can read the [optimization walkthrough](#) for a look at applying the optimizations in a real program.

Algorithmic Optimization

An algorithm refers to your method of solving a problem. Algorithmic optimization, then, relies on choosing the best method to solve a particular problem. Unlike line-by-line optimization, even a single optimization of this type can have drastic results — but it also requires critical thinking and a case-by-case approach.

Most programmers, after thinking about the methods they will use for a while, never spend much time on this kind of optimization. It becomes important when you're pushed in a corner: your program has become so large that it doesn't have enough memory to run, or takes half a minute to load each screen.

Identify the bottleneck in your program — what is it that takes up all the memory, or that the program spends so much time doing? Then consider several fundamentally different approaches to solving that particular problem (be it the problem of storing a large matrix or of displaying a tilemap). Write routines implementing each approach, fully optimize all of them, and compare the results. And make sure that you're not missing an approach too radical to think of. Virtually all of the techniques you find in this guide have been discovered by frustrated programmers doing exactly this kind of thinking.

Read the [algorithmic optimization](#) tutorial for a demonstration of the process of algorithmic optimization in a real programming situation.

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/optimize>

Code Timings

This page documents the speed of certain commands in TI-83 Basic. Although the times given

here will vary from model to model and even from calculator to calculator (due to battery levels, free memory, and other factors), one thing that does not change is the relative speed of the commands. So, if you come here to see if a For(loop is faster than a While loop, the information will be useful on any calculator.

Elsewhere on this site, you might see assertions like "foo() is faster than bar()" without any reason or proof. The information on this page is the reason and proof behind them.

fold

Table of Contents

[Testing Format](#)
[Contributing your own Tests](#)
[Program Main Code](#)
 [If statements](#)
 [Relational Operators](#)
 [int\(vs. iPart\(](#)
 [The getKey Function](#)
 [For\(, Repeat and While Loops](#)
[Graphing Code](#)
 [The pxl-Test\(Function](#)
 [Pixel and Point Modifying](#)
 [Text\(vs. Output\(vs. Disp\(](#)
[Optimizing your Code](#)
 [Parentheses and Quotes](#)
 [Multiplication, Division and Addition](#)
 [Using Ans](#)
 [Using Finance Variables](#)
 [Recalling Lists](#)
 [Imaginary vs. Decimals](#)
 [Calculating powers of 10](#)
 [IS>\(vs. If command](#)
 [Alternate methods](#)
 [getKey routines](#)

Testing Format

In order to be able to compare speed results between commands, there needs to be a common format that is used for all of the tests. However, there are actually two different formats that you can use depending on which TI-83 based calculator you have.

The first format is for those with a TI-83, TI-83+, or TI-83+SE, and it is just a simple For(loop that is executed a set number of times over the command:

```
:For(A,1,(number)
: <command(s) to be tested>
:End
```

You measure the time by getting out a stopwatch, and trying to estimate the number of times the run indicator moved. The run indicator is the little, one pixel wide bar in the upper right corner of the calculator that moves when you run a program. Each completed run indicator you count as eight, and then any leftover pixels you simply add to the total.

Of course, because the run indicator moves quite fast, this testing format can be plagued by human error. If you have any second guessing or are unsure if a timing was correct, you should run the test again. You can then take the average of the two times as your result.

The second format is for those with a TI-84+ or TI-84+SE, and it involves using the built-in `startTmr` and `checkTmr(` commands. You first store `startTmr` to a variable (usually a real variable), and then run your command inside of a `For(` loop. You then check the time with the `checkTmr(` command using the variable from before that `startTmr` was stored to.

Here is a standard template to use:

```
:ClockOn  
:startTmr→T  
:For(A,1,(number)  
: <command(s) to be tested>  
:End  
:checkTmr(T)/(number)
```

Making (number) higher increases accuracy, but takes longer. Also, make sure not to modify the variables A or T inside of the `For(` loop.

While this format eliminates human error from counting, it's prone to its own faults. A major one is that `startTmr` and `checkTmr(` always return whole numbers, but time is continuous. Depending on how close the start and end of the loop were to a clock tick, the number of seconds may be off by up to one second in either direction. To take this into account, you could replace the last line:

```
: (checkTmr(T)+{-1,1})/(number)
```

This will give you a list of the maximum and minimum possible times — the true time that the command takes is guaranteed to be somewhere in between.

If there is a need to use one or more variables during a test, you should initialize the variables to a known value before running the test. You can do this either on the home screen or before the format code (in which case, you should also put a Pause to separate the variable initialization from the code test).

Contributing your own Tests

Feel free to experiment with code timings, and to put your results up on this page. However, be sure to list the calculator model and the OS version (found in the About menu) that you used! Unless stated otherwise, all tests on this page were done with a TI-83+ and OS version 1.19.

That's it for details and explanations. Now come the actual timings!

Program Main Code

If statements

This very first section is a difficult one to approach, because the nature of our testing method affects it. It turns out that the `For(` command, when it doesn't have a closing parenthesis after it, will slow down If statements with a false condition inside the loop. This doesn't affect the speed of any other commands (except `IS>(` and `DS<(` which are rarely used), nor does this effect

occur with a true condition, nor with If-Then-End blocks (just with a single If and a command following it). These two pieces of code will be affected, for instance (the second will be much faster):

<pre>:For(I,1,100 :If 0: :End</pre>	<pre>:For(I,1,100) :If 0: :End</pre>
-------------------------------------	--------------------------------------

The following table summarizes all of these effects. It would have been too cumbersome to maintain the same format as elsewhere, so the number is simply the total number of pixels.

Conditional type	For(A,0,2000)	For(A,0,2000)
If 0:	1520	79
If 1:	79	82
If 0:Then:End	80	83
If 1:Then:End	89	91

Conclusion: The ending parenthesis situation, when it **is** applicable, is a major factor, slowing the statement down nearly 20 times. For this reason, I suggest that if there's any chance at all the condition is false (which is always the case, or else why are you testing for it in the first place?) to leave off the parenthesis on the For(loop. Of course, this doesn't affect If-Then-End commands.

It was long held, because of a misunderstanding of this effect, that If commands were slower than If-Then-End (prior versions of this page were not entirely innocent, either). As you can see, this is not the case, as long as you are aware of the effect shown above. Though there are slight differences in the timings, they are so small that you can ignore them.

Relational Operators

In all programs, there is a lot of chances that you will see relational operators used to determine what to do. But will each of them take the exact same time to work?

Format	Bars	Pixels	Total
=	10	4	84
≠	10	5	85
>	10	5	85
≥	10	5	85
<	10	5	85
≤	10	6	86
and	10	6	86
or	10	6	86
not(8	6	70
xor	10	5	85

Conclusion: If you can reverse your operations, then do it to save some time.

int(vs. iPart()

As you may know, int(and iPart(have the same use, for positive numbers at least.. In programs where you store more than one variable in 1 number, you normally use int(or iPart(, but which is the best one to use?

Format	Bars	Pixels	Total
iPart(1	10	1	81
iPart(1.643759	10	1	81
int(1	8	7	71
int(1.643759	10	2	82

Conclusion: Unless there are 6 or more decimals, you should consider using int(because of it's speed, but with several decimals, iPart(stays the same so it goes faster.

The getKey Function

I hope you all know the getKey function, it is probably the most used in games and custom menus. It takes time, but we still don't know if it is fast...

Format	Bars	Pixels	Total
Getkey	7	5	61
Getkey→B	11	0	88

Conclusion: getKey is pretty fast, but storing to the variable takes a lot of time. So, if you don't need to have the value of the key pressed, don't store it and use the special variable Ans instead.

For(, Repeat and While Loops

There are many types of loops that you should know already: For(, Repeat and While loop. But if we have the choice, which one is faster?

```
:For (A, 0, 2000  
:While 0 :End  
:End
```

12 bars +2 pixels (98 pixels)

Also see that with For(A,B,C loops implementation, you can do an If statement:

If B≤C, then C+1→C (at the end)
If B>C, store B into A

```
:For (A, 0, 2000  
:For (B, 1, 0  
:End  
:End
```

12 bars (96 pixels)

```
:For(A,0,2000  
:Repeat 1  
:End  
:End
```

13 bars +2 pixels (106 pixels)

Conclusion: There is one loop that is best. Use the right loop for the task you need to do.

```
:For(A,0,2000  
:End
```

4 bars +4 pixels (36 pixels)

```
:Delvar A  
:While A≤2000  
:A+1→A ;No use of Ans because there should be other code in the lo  
:End
```

23 bars (184 pixels)

```
:Delvar A  
:Repeat A>2000  
:A+1→A ;No use of Ans because there should be other code in the lo  
:End
```

22 bars +7 pixels (183 pixels)

Conclusion: For the same use, please use a For(loop...

Graphing Code

The pxl-Test(Function

Many TI-BASIC programmers reported issues of when pxl-Test(is a conditional, it takes up to 40% more time.

Format	Bars	Pixels	Total
pxl-Test(15,15 ;pixel turned off	12	1	97
pxl-Test(15,15 ;pixel turned on	12	1	97
If pxl-Test(15,15: Then: (empty line): End ;pixel turned on	20	0	160
If pxl-Test(15,15: Then: (empty line): End ;pixel turned off	18	6	150

pxl-Test(15,15: Then: (empty line): End ;pixel turned on	24	2	194
pxl-Test(15,15: Then: (empty line): End ;pixel turned off	22	7	183

Conclusion: For my calculator, at least, it didn't give me the errors reported by others. So don't use pxl-Test(:If Ans, but If pxl-Test(), it goes faster and takes a byte less. Also, it doesn't matter whether pixel is on or off.

Pixel and Point Modifying

The objective in having games on the 83+ is mostly because it has good graphics that are entertaining. This is why we need to open or close pixels in order to draw. I made my test with a window size of: Xmin=0, Ymin=-62, Ymax=0, Xmax=94

Format	Bars	Pixels	Total
Pt-On(15,-15	14	0	112
Pt-On(15,-15,2	20	1	161
Pt-On(15,-15,3	18	2	146
Pt-Off(15,-15	14	0	112
Pt-Off(15,-15,2	20	1	161
Pt-Off(15,-15,3	18	2	146

Conclusion: So like we see, Pt-On/Off is the same time of execution.

Format	Bars	Pixels	Total
Pt-Change(15,-15	14	0	112
Pxl-On(15,15	9	4	76
Pxl-Off(15,15	9	4	76
Pxl-Change(15,15	9	4	76
Line(15,-15,16,-15	16	2	130
Line(15,-15,30,-15	32	6	262
Line(15,-15,30,-15,0	34	6	178
Horizontal -15	82	5	661
Vertical 15	60	3	483

Conclusion: Line(), Horizontal and Vertical are all slow, but they can save bytes. If there are under 4 or 5 pixels to turn on, Pxl-On() works much faster than any of them. However, if you have a lot of pixels to turn on/off, it is much better to use them than the Pxl-/Pt- commands. Also, Pt-Change() is the same speed wise as Pt-On/Off.

Text(vs. Output(vs. Disp

In all your programs, there is probably something that displays text on the screen. There are many ways to do so, so I will look at them to see which one is faster. The codes will be displaying the same string, "I DIE!", so that I can give you valuable timings. In order to find the timing of display, ClrHome is after all of the commands.

Format	Bars	Pixels	Total
Text("I DIE!")	10	0	10

Text(-1,16,12,"I DIE!")	54	4	436
Text(16,12,"I DIE!")	41	1	329
Output(3,2,"I DIE!")	37	6	302
Disp "I DIE!"	51	2	410

Conclusion: For immobile text, if you need to be big, you should use Output(), but if you need it into graph screen, then think about the time it takes...

Format	Bars	Pixels	Total
Output(3,2,A)	16	6	134
Text(-1,16,12,A)	36	7	295
Text16,12,A	27	0	216

Conclusion: For variables' values, same thing applies.

Optimizing your Code

Parentheses and Quotes

Normally, you shouldn't close parentheses and quotation marks to save a byte. I will test if it goes faster.

Format	Bars	Pixels	Total
Output(3,2,"I DIE!")	20	6	166
Output(3,2,"I DIE!"	20	6	166
(5+6)→B	13	6	110
(5+6→B	13	5	109
5+6→B	13	2	106

Conclusion: The only reason you need to get out the quotations marks are because you save 1 byte, you don't get faster. Also, taking off closing parenthesis goes faster. However, it is better if you can get rid of the parentheses entirely.

Multiplication, Division and Addition

Most TI-BASIC programmers tell you not to put the multiplying * sign, but do they know if it goes faster?

Format	Bars	Pixels	Total
A*B	13	4	108
AB	13	2	106

Conclusion: If you multiply, don't put the * sign.

Format	Bars	Pixels	Total
If AB: Then: (empty line): End ;Condition true	20	5	165
If A and B: Then: (empty line): End ;Condition true	20	5	165

If AB: Then: (empty line): End ;Condition false	18	7	151
If A and B: Then: (empty line): End ;Condition false	19	2	154

Then you could possibly use the AB format because there is 1 byte less and no speed loss and if the condition is mostly false, AB goes faster...

Format	Bars	Pixels	Total
If C+B: Then: (empty line): End ;Condition true	20	6	166
If C or B: Then: (empty line): End ;Condition true	20	6	166
If C+B: Then: (empty line): End ;Condition false	19	3	155
If C or B: Then: (empty line): End ;Condition false	19	7	159

Same as for the last tests, but you don't save any space.

Format	Bars	Pixels	Total
If A(C+B: Then: (empty line): End ;Condition true	25	4	204
If A and (C or B: Then: (empty line): End ;Condition true	25	1	201
If A(C+B: Then: (empty line): End ;Condition false	23	4	188
If A and (C or B: Then: (empty line): End ;Condition false	23	7	191

Conclusion: So as we can see, in multiple conditions, where it should be true a lot of time, you should use the *and* and *or* operators instead of multiplication and addition.

Format	Bars	Pixels	Total
A/B	20	6	166
AB^-1	28	2	226

The following timings were taken on a TI-84+ SE with OS version 2.40

Format	Bars	Pixels	Total
1/B, when B=1	15	0	120
B^-1, when B=1	14	1	113
1/B, when B=pi	20	2	162
B^-1, when B=pi	19	2	154

Conclusion: When dividing two numbers, don't use the $\wedge -1$ operation. It goes really slow! But if you're only taking an inverse, use the $\wedge -1$ operation instead of dividing from 1.

Using Ans

Normally, you try that your program goes faster by optimizing it. Maybe you know, maybe you don't, but using the special variable Ans is supposedly faster than normal variables like A, B, C, θ, and others. In this test, B is starting with a value of 1.

```
:For (A ,0 ,2000
:Ans+1→B
```

```
:For (A ,0 ,2000
:B+1→B
```

: End

: End

14 bars +6 pixels (118 pixels)

15 bars +1 pixel (121 pixels)

Conclusion: Using Ans is a little faster than using the real variables.

Using Finance Variables

Have you ever heard of Finance variables other than in this guide? Probably not, they are more space taking than other variables. Let's see if there is any advantage of using them.

Format	Bars	Pixels	Total
N+1→N	11	5	93
Ans+1→N	13	0	104
N	8	0	64
Ans	9	3	75

Conclusion: Use Finance Vars! Although, it is your choice: a 2 byte, really fast variable (i.e., the finance variables) or a 1 byte, slow variable (i.e., the real variables).

Recalling Lists

As you know, lists are arrays of variables, that you can modify specifically, one by one. You can use pre-defined lists, such as L₁, L₂, ..., L₆, and user-defined lists, LXXXXX where X represents any letter or number or nothing at all (except for the first character). It takes time recalling an element, but how much?

Format	Bars	Pixels	Total
\L1\	13	6	110
\L1\1	15	0	120
\L\A	15	3	123
\L\A(1	16	6	134
\L\AA	15	5	125
\L\AAA	15	4	124
\L\AAAA	16	2	130
\L\AAAAA	16	2	130
A	9	6	78

Conclusion: If you can, use pre-defined lists as temporary buffer, but not for long-term storage, it is so easy to get it modified in a math class. And if you can, use real variables instead of lists if you have very few elements and that the data storage is not long-term.

Imaginary vs. Decimals

If you have looked in some tutorials, they talk about having many different variables held in one variable, by using either imaginary numbers in rectangular form (A+Bi), or decimal points (XX.YYMMDDNNIIJJ). It saves space for keeping track of saved games, and sometimes time if

you use it correctly. But which ways are the fastest?

Format	Bars	Pixels	Total
real(4+4i)+imag(4+4i)	23	4	188
real(4+4i)	14	6	112
imag(4+4i)	14	6	112
int(4.4)+10fPart(4.4)	15	2	116
int(4.4)	8	6	70
10fPart(4.4)	11	1	89

Conclusion: If you can, try not to use imaginary rectangles, they are slower than their int() and fPart() equivalent and they store the exact same amount of data. Besides that, a complex variable is twice as big as a real variable, and if you use one in a list it will make even the real elements twice as big.

Calculating powers of 10

The calculator has at least three ways to calculate some power of 10: using the small E command (limited to integer powers), using the 10^(command, and typing out 10^{\wedge} . How do these compare?

Format	Bars	Pixels	Total
E1	6	6	54
$10^{\wedge}1$	9	0	72
$10^{(1)}$	12	4	100
E99	6	7	55
$10^{\wedge}99$	50	6	406
$10^{(99)}$	12	6	102

Conclusion: The E command wins out by far, but it's limited, so you can't always use it. In those cases, typing out 10^{\wedge} is slightly faster than the 10^(for small arguments (the breaking-even point seems to be around $10^{\wedge}9$), but is a lot slower for large arguments. Of course, there's also the size to consider, so the command seems to be a pretty safe bet.

IS>() vs. If command

This is what happened when I compared IS>() to If conditionals:

Format	Bars	Pixels	Total
IS>(B,10):Disp	24	2	194
If B>10:B+1→B:Disp	34	3	275

Conclusion: IS>() works faster, but its flaws might not make it very useful.

Alternate methods

getKey routines

These are two different methods of moving an X on the homescreen.
Darkstone Knight's alternate method. (123 bytes, 12 bars, 7 pixels, 103 pixels total)

```
:ClrHome  
:1.01→A  
:For(D,0,200)  
:getKey  
:If Ans  
:Output(iPart(A),(smallcapitalE)2fPart(A)," /one space/  
:A+(Ans=34 and A<8)-(Ans=25 and A≥2)+sub((Ans=26 and fPart(A)<.16)-  
:Output(iPart(A),(smallcapitalE)2fPart(A),"X  
:End
```

and the original method using piecewise expressions. (109 bytes, 13 bars, 104 pixels total)

```
:ClrHome  
:1→A  
:1→B  
:For(D,0,200)  
:getKey→C  
:If C  
:Output(A,B," /one space/  
:A+(C=34 and A<8)-(C=25 and A>1→A  
:B+(C=26 and B<16)-(C=24 and B>1→B  
:Output(A,B,"X  
End
```

So which one you use depends on your value of 1/200th of a pixel per iteration vs. 14 bytes of size.

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/timings>

Releasing Your Program

Many programming guides give you excellent advice on programming, but stop at the point when the program is finished, tested, and optimized. After all, most people can manage to release a program somewhere, one way or another. But in reality, an inexperienced programmer may well release his work quietly and in an unassuming form, which people will simply glance over without stopping. This tutorial will tell you how to avoid this, and make your program get all the attention it deserves.

Where to Release

First, it's important to know where to go to upload your program to the Internet. Although you might want to create your own website and release all your games there, that alone will not get your program noticed. Sure, having your own site might get you some publicity, but the best way to get your game noticed is by releasing it at one (or all!) of the large program archives.

- ticalc.org

- [CalcGames.org](#)
- [United-TI](#)
- [TI-Basic Developer](#)

Of these, ticalc.org is by far the largest (and most popular), but it's also likely you'll spend longer waiting for your program to be put up there. With CalcGames, and United-TI, you only have to wait a day or two. With TI-Basic Devloper, you only have to wait a few minutes, or you could do it yourself.

What to Release

There's more you'll want to submit than just the program itself. Here are the elements you'll want to put together — some of these are called optional by the file archive websites, but they are mandatory if you want the program to be successful.

The program itself (obviously)

If you were programming on the calculator, you'll need to transfer the program to your computer to submit it. You'll need a calculator-to-computer cable, and software such as TI-Connect. If you don't know where to get these, or have problems using them, see [linking](#).

Now, you have one or more files from your calculator on the computer. If there's only one, you're good to go. If there are several files involved, you should consider combining them in a [group](#) file (usually .83g or .8xg). Or keep them like they are, but then make sure to mention what each file is for, in the readme.

Although, if you don't want to worry about having to ungroup, or group the files, another option is [Basic Builder](#). Basic Builder packages your programs, in an app. More information, is given at [this page](#).

The readme

A critical step in submitting a program. Make sure to read our tutorial on [writing a readme](#) if you've never done it before (and possibly even if you have). Usually, longer is better than shorter (it's worse if someone doesn't understand how your program works, than if they have what they already know explained to them again) — unless it's a five-act play, in which you might consider removing the nonessentials. Generally, the longer and better your program, the longer your readme can be; you don't need any more than the minimum for, say, a quadratic solver. For a huge program, a 2-4 page plain text file is appropriate.

Also, please don't make the readmes in Microsoft Word 7 file format! A .txt file is sufficient, and in fact recommended. However, if you're just itching to put screenshots, pictures, and format your whole paragraph accordingly, a .pdf file would be a good idea. PDF files can be read by most computers automatically, but if not, Adobe reader, is free. It might be a good idea, to put a file with a link to an adobe download station. Most likely <http://get.adobe.com/reader/> will be the link to get adobe reader. You might also want to mention that it's free. Make sure you have that .txt file that gives the information on where to find adobe.

The screenshot

All four websites listed above let you add a still or animated screenshot of your program. This is very easy to do — see the [making a screenshot](#) page — and goes a long way toward making your program look good (if it actually is good). An attractive screenshot will encourage visitors to download your program more than the most flowery prose. Show your program at its most impressive here.

Getting a screenshot is easy, open TI Connect, in 1.7 and 1.6, it should look like a camera. Click it.

The title

The title will tell visitors what your program is all about. One common mistake is making the title the same as the 8-character name of the program. Don't do this — the title is the first thing people will see, and you want to make it clear. Of course, if the program is called prgmTETRIS it's okay to call it Tetris (though Grayscale Tetris, if that's the case, could be even better). But if the program is called prgmQUADSOLV, *please* make the title Quadratic Solver instead!

The description

Don't forget this! It should have three parts:

- What the program is about. "Solves all quadratic equations over the complex numbers."
- The program's best qualities. "A grayscale interface at the low size of 13 bytes!"
- Any requirements. "Requires xLIB, Omnicalc, Symbolic, and DAWG to work correctly. Also, create and unarchive GDB7."

The first two parts are positive; the third is negative, but necessary (imagine if your program crashes without warning if GDB7 is not created. 99% of your users will be lost, even if this is explained in the readme, and write negative reviews). You want to make this section as short as possible, and the best way to do this is to avoid the requirements in the first place. Even if your game is in the "games for xLib" category, the one who is looking for a game might not see this, and not download, or install xlib.

Putting this together

The program and the readme should be combined in a .zip archive, this is a community-wide standard. The file upload form (this is different for all websites, but contains the same basic information to be entered) should have fields where you can submit everything else. You might also consider adding the screenshot to the .zip archive, **in addition** to its normal location.

Here are the links to the file upload forms of all the websites mentioned on this page.

- [Ticalc.org's form](#)
- [United-TI's form](#)
- [CalcGames.org's form](#)
- [TI-Basic Developer's form](#)

Note: You need to create an account at the respective website before you can upload files there.

Marketing

Marketing your program can start as early as when you first get the idea for your program, although many people won't take you seriously until you have at least a basic engine to show for your efforts. Other good points at which to advertise the program include a beta-testing period before you release it to the masses, and of course when it's finally released. For more marketing tips, see our [marketing](#) tutorial.

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/releasing-your-program>

Optimization: Conditionals

Use If conditionals when you only want to execute the one command on the next line.

```
: If A=1  
: Then  
: C+2→C  
: End  
can be  
: If A=1  
: C+2→C
```

Because conditionals are generally slow, you should replace them with piecewise expressions if you are just changing a variable. You take the variable and add or subtract the expression, multiplying it by the value that you are adding to the variable. Using piecewise expressions can sometimes be slower than If conditionals to avoid storing zero into the variable if the expression is false.

```
: If A=3  
: B+2→B  
can be  
: B+2 (A=3→B
```

You don't need to put the value in front of the expression when it is one.

```
: B+1 (A=2→B  
can be  
: B+ (A=2→B
```

You can take piecewise expressions a step further by combining multiple If conditionals that deal with the same variable and put them into one piecewise expression.

```
: If A=3  
: B+5→B  
: If A=6  
: B - 3→B  
can be  
: B+5 (A=3) - 3 (A=6→B
```

If you are adding and subtracting the same value from the variable in the piecewise expression, you can factor the common value from each expression. This works best when you are adding and subtracting a big number.

```
: B+11 (A=1) - 11 (A=2→B  
can be  
: B+11 ((A=1) - (A=2→B
```

You can sometimes reorder a list of If conditionals so that the last possible outcome doesn't even need an If conditional. This mainly works when the program is going to do a certain action and there are no other alternative actions that can occur.

```
: If not(A
```

```
:Goto A
:If A=1
:Goto B
:If A=2
:Goto C
can be
:If A=2
:Goto C
:If A
:Goto B
:Goto A
```

If-Then-End conditionals should be used when you want to execute multiple commands.

```
:If A=1
:C+1→C
:If A=1
:D+1→D
can be
:If A=1
:Then
:C+1→C
:D+1→D
:End
```

If you have two or more If conditionals that have a common expression, you should take the common expression out and make it into an If-Then-End conditional and nest the If conditionals inside it.

```
:If A=1 and B=1
:C+2→C
:If A=1 and B=2
:D+1→D
can be
:If A=1
:Then
:C+2(B=1→C
:D+(B=2→D
:End
```

If you are displaying lots of text based on If conditionals, you should put the text together and then just use the sub command to get the appropriate part of the text. This will display the text if none of the conditions are true, so this may not always be desired.

```
:If A=3
:Disp "Hello
:If A=4
:Disp "World
can be
:Disp sub("HelloWorld",1+5(A=4),5
```

The If-Then-Else-End conditionals should be used if you want to execute multiple commands

when an expression is true or false. Instead of putting two If-Then-End conditionals that have math opposite expressions, If-Then-Else-End conditionals are faster because you don't need to do two checks; only one of the conditionals can be true at one time.

```
:If B
:Then
:"Hello→Str1
:End
:If not(B
:Then
:"Goodbye→Str1
:End
can be
:If B
:Then
:"Hello→Str1
:Else
:"Goodbye→Str1
:End
```

When using an If-Then-Else conditional and only one command is executed if the expression is true or false, use an If conditional between the two commands instead. You might also have to change the order of the commands, depending upon the commands.

```
:If B
:Then
:"Hello→Str1
:Else
:"Goodbye→Str1
:End
can be
:"Goodbye→Str1
:If B
:"Hello→Str1
```

When a line is either drawn or erased depending on a condition, you can put that condition as the optional fifth argument for the Line command.

```
:If B:Then
:Line(1,2,3,4
:Else
:Line(1,2,3,4,0
:End
can be
:Line(1,2,3,4,B
```

When you have a If-Then or If-Then-Else conditional that has a Goto command as one of the nested commands, you can sometimes remove the conditional and replace it with multiple If conditionals. Doing this prevents a memory leak from happening.

```
:If A
:Then
```

```
:Disp "Hello
:Goto A
:Else
:Disp "Goodbye
:B+2→B
:End
can be
:If A
:Disp "Hello
:If A
:Goto A
:Disp "Goodbye
:B+2→B
```

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/optimize-conditions>

Optimization: Deleting Variables

Instead of setting number variables to zero (to delete them), use the DelVar command. DelVar works with all of the variables, and the calculator automatically sets the variable to zero the next time it's used.

```
:0→A
can be
:DelVar A
```

The DelVar command doesn't need a line break or colon following the variable name. This allows you to make chains of variables.

```
:DelVar A
:DelVar B
can be
:DelVar ADelVar B
```

Besides making chains of variables, the DelVar command also allows you to take the command from the next line and put it immediately after the last DelVar command.

```
:DelVar A
:Disp "Hello
can be
:DelVar ADisp "Hello
```

The only exception is with the Lbl command. Don't put the Lbl command immediately after a DelVar with this optimization, or else the label will be ignored. For instance, the following code exits with ERR:LABEL:

```
:DelVar ALbl 0  
:Goto 0
```

Even though the [ClrList](#) command exists for clearing lists, DelVar should be used instead.

```
:ClrList L1  
can be  
:DelVar L1
```

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/optimize-deleting>

Optimization: Exiting Programs

Although the Return and Stop commands can both be used for exiting programs, Return should be used instead of Stop. While Return stops only the current program and allows the parent program to continue running, Stop causes all of the programs to stop and then returns the user to the homescreen (unless called from an Assembly program).

```
:ClrHome  
:Disp "Hello  
:Stop  
can be  
:ClrHome  
:Disp "Hello  
:Return
```

You don't have to use Return or Stop if you can organize the program so that it just naturally quits. If the calculator reaches the end of a program, it will automatically stop executing.

```
:ClrHome  
:Disp "Hello  
:Return  
can be  
:ClrHome  
:Disp "Hello
```

When you have a display command that displays text as the last line of the program, you can remove the command and just put the text. This text will be displayed instead of the "Done" message that is normally displayed after a program finishes executing.

```
:ClrHome  
:Disp "Hello  
can be  
:ClrHome :"Hello
```

Even though you don't display any text as the last command, you may still want to get rid of the

"Done" message. You can do this by putting a single double-quote as the last line of the program.

```
:ClrHome  
can be  
:ClrHome:"
```

If you modify the Ans variable on the last line of the program, Ans's new value will be displayed instead of the "Done" message.

```
:ClrHome  
:For(A,1,5  
:B+A→B  
:End  
can be  
:ClrHome  
:For(A,1,5  
:B+A→B  
:End  
:B
```

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/optimize-exiting>.

Optimization: Logic and Relational Operators

Because the calculator treats every nonzero value as true and zero as false, you don't need to compare if a variable's value is nonzero. Instead, you can just put the variable by itself.

```
:If C≠0  
can be  
:If C
```

Instead of comparing a variable to zero, use the not logical operator. Because not returns the opposite value of the variable, true will become false and false will become true.

```
:While A=0  
can be  
:While not(A
```

When making expressions that combine the and and or operators where the and operator comes first, you don't need to include parentheses around the and operator. The and operator has a higher precedence than or, so it is evaluated first. This can become complicated with complex expressions, so you might want to leave some of the parentheses for clarity.

```
:If (A=1 and B=2) or (A=2 and B=1)
can be
:If A=1 and B=2 or A=2 and B=1
```

If you are comparing two unary expressions (expressions with no comparison operator) with the and operator, you don't need the and operator. For and to be true, both values must be nonzero. So, multiplying them will produce the same effect because if either one of the values is zero, the product of both values will also be zero.

```
:If A and B
can be
:If AB
```

A similar technique can be applied to expressions with comparison operators, except some restrictions are required.

With unary expressions, to test if A and B is true you multiply them. With equations, you can multiply the left sides of each together and you can do the same for the right sides. However, a value being 0 could return a different result than anticipated, so it is best to use this technique when the values are not 0.

```
:If A=B and C=D
can be
:If AC=BD
```

As and is similar to multiplying, the or operator is similar to addition. Adding two values together yields a non-zero result if one of the conditions is true. When you are comparing equations using the or operator, you can add the two together (This is not used for unary expressions because the plus symbol and or symbols are both one-byte tokens). For this the only restriction is that all values must have the same sign (or be 0), or you can circumvent this by using abs. This is necessary because if two variables have the same value except one is negative, this expression could return false.

```
:If A=B or C=D
can be
:If A+C=B+D
```

The most unused logical operator is xor (exclusive or). The xor operator is useful when comparing two expressions and checking if one but not both are true. In fact, xor is specifically designed for this purpose.

```
:If A=2 and B≠2 or A≠2 and B=2
can be
:If A=2 xor B=2
```

Many times a compound expression can be shortened by combining expressions that have the same meaning or replacing expressions that can be written another way. Think about what the expression means and then think about how to make a shorter equivalent expression. There are many ways of writing an expression, so there are usually ways to rewrite it.

```
:If A>B or A<B  
can be  
:If A≠B
```

If you have the not operator around an expression, you can usually change the logical operator to the math opposite. This allows you to remove the not operator.

```
:If not(B=C and A=D  
can be  
:If B≠C or A≠D
```

DeMorgan's Law can be used for expressions in which the not operator is around two separate unary expressions joined by the and or or operators. It allows you to remove the second not operator and then change the and to or and vice versa.

```
:If not(A) and not(B  
can be  
:If not(A or B
```

Min is useful when you are comparing one variable or value to several other variables to see if they are all equal to the variable or value. To use min you just create an expression with the min function and put the common variable or value inside it followed by an equal sign and a left curly brace. You then list out the variables that you are comparing the variable or value to, separating each one with a comma.

```
:If A=10 and B=10 and C=10  
can be  
:If min(10={A,B,C
```

Max is useful when you are comparing one variable or value to several other variables to see if at least one is equal to the variable or value. You do the same thing as the min function, just replacing min with max.

```
:If A=10 or B=10 or C=10  
can be  
:If max(10={A,B,C
```

You can put a comparison operator inside the min or max functions to compare when several values or variables are equal to one variable and several values or variables are equal to another variable. This works especially well with three or more variables.

```
:If A=X and B=U or A=Y and B=V  
can be  
:If max(A={X,Y} and B={U,V
```

Abs is useful when you are comparing a variable to two even or odd values using the or operator. You subtract the larger value from the smaller value, divide the result by two, and then put it on the left side of the equal sign. Next, you subtract the larger value by the result on the

left side of the equal sign, and then take the variable being tested and subtract it by that value. You then put the abs function around the result and place the expression on the right side of the equal sign.

```
:If A=45 or A=105  
can be  
:If 30=abs(A-75
```

X=n1 or X=n2 should become $\text{abs}(\text{n1}-\text{mean}(\{\text{n1}, \text{n2}\}))=\text{abs}(X-\text{mean}(\{\text{n1}, \text{n2}\}))$ (simplified) if n1 and n2 are positive integers and n1+n2 is even. If there are three terms, then see if you can simplify two of them according to this rule. If you can't, then a string of or's will be faster than the $\text{max}(X=\{\text{n1}, \text{n2}, \dots\})$ approach. If there are four terms or more, then use $\text{max}()$.

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/optimize-logic>

Optimization: Loops and Branching

When using loops you want to make them as compact as possible. This starts with moving invariant code outside the loops. You only want loops to contain expressions whose values change within the loops. If something only happens once, it should be outside the loop.

```
:For(X,1,5  
:5→Y  
:Disp X  
:End  
can be  
:5→Y  
:For(X,1,5  
:Disp X  
:End
```

You also want to minimize the calculations inside loops. This not only includes cutting down on the number of storages, but how often variables are used and what they are used for. This can increase the size, however.

```
:For(X,1,10  
:A+length(Str1→A  
:End  
can be  
:length(Str1→B  
:For(X,1,10  
:A+B→A  
:End
```

Another way to minimize calculations inside loops is to use constant increments. This makes the loop faster, but it also makes it larger.

```
:For(X,0,10
:Disp 10X
:End
can be
:For(X,0,100,10
:Disp X
:End
```

You should combine two or more loops that are in close proximity if they use the same number of iterations and don't affect each other. Combining loops may take some ingenuity.

```
:For(X,1,10
:B+X→B
:End
:For(Y,1,10
:A+A/Y→A
:End
can be
:For(X,1,10
:B+X→B
:A+A/X→A
:End
```

Loop unrolling reduces the number of times you check the condition in a loop, with two or more of the same statements being executed for each iteration. If the loop is small enough, you can even unroll the whole loop. This will usually increase the size but also make it faster.

```
:5→dim(L1
:For(X,1,5
:2A→L1(X
:End
can be
:5→dim(L1
:2A→L1(1
:2A→L1(2
:2A→L1(3
:2A→L1(4
:2A→L1(5
```

For(loops are best used when you know how many times the loop will be executed. Because the fourth argument is optional (one is the default), you should always try to leave it off.

```
:For(X,1,8,1
:End
can be
:For(X,1,8
:End
```

You can sometimes rewrite For(loops and the commands inside them so you can remove the fourth argument.

```
:For(X,8,0,-1
:Disp X
:End
can be
:For(X,0,8
:Disp 8-X
:End
```

If you have an If conditional around the outside of a For(loop, you should see if there is a way to combine it with the For(loop using Boolean logic.

```
:If A>10:Then
:For(X,1,50
:End:End
can be
:For(X,1,50(A>10
:End
```

One of the common uses of For(loops is to slow programs down. Instead of For(loops, you should use rand(# or If dim(rand(#). Both of these create lists of random numbers, with a larger number meaning a larger delay; the second one preserves the Ans variable as well.

```
:For(X,1,75
:End
can be
:rand(25
```

This method generally works well for small delays, but it is better to use For(loops for large delays. This is because the rand(# technique is limited by the RAM storage availability, and has a maximum delay of 999 (being a list variable).

```
:rand(200
can be
:For(X,1,600
:End
```

Repeat loops will loop until the expression is true, and While loops will loop while the expression is true. Repeat loops are tested at the end of the loop which means they will be executed at least once. This allows you to not always have to set the variables in the expressions, which is the case with While loops. If the expression in a While loop is false before it is tested, the loop will be skipped over. This is sometimes desired if the expression fits that format.

```
:DelVar A
:While not(A
:getKey→A
:End
can be
:Repeat A
:getKey→A
```

```
: End
```

If you need a loop that loops forever (i.e., an infinite loop), use Repeat 0 or While 1 instead of Goto/Lbl.

```
:Lbl A
:Disp "Hello
:Goto A
can be
:Repeat 0
:Disp "Hello
:End
```

Goto/Lbl loops should be used sparingly. When Goto is encountered, it notes the Lbl and proceeds to search for it from top to bottom in the code. This can really be slow if the Lbl is deep within the program. It also has the tendency to make your code harder to follow and maintain. And, if you use a Goto to exit a loop or a conditional that uses an End command, it can lead to memory leaks (causing your program to crash).

```
:Repeat 0
:getKey→B
:If B
:Goto A
:End
:Lbl A
can be
:Repeat B
:getKey→B
:End
```

When all a For(loop does is store expressions to a list, you can replace it with a seq((sequence) command. The sequence command can also be used with other variables.

```
:5→dim(L1
:For(X,1,5
:2A→L1(X
:End
can be
:seq(2A,X,1,5→L1
```

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/optimize-loops>

Optimization: Math Operations and Keys

Multiplication signs are unnecessary and should be removed because the calculator does

implicit multiplication. You should remember that implicit multiplication doesn't bind tighter than regular multiplication.

```
: 5 * A→B  
can be  
: 5A→B
```

You don't need to put parentheses around a single variable or number by itself when doing multiplication or division.

```
: 3 / (A)  
can be  
: 3/A
```

Multiplication and division have the same importance based on the order of operations (the rules that determine what order things are evaluated in), so they will be evaluated from left to right if both appear in an expression. If multiplication appears before division, you can remove the parentheses around an expression.

```
: A+ (BA) / 5→C  
can be  
: A+BA/5→C
```

Although multiplication and division have the same importance in order of operations, multiplication is in fact faster than division when doing math operations. So, you should multiply instead of dividing, especially if doing the multiplication is smaller than doing the division.

```
: (X+1) / 2  
: (B+C) / D  
can be  
: . 5 (X+1  
: D¹ (B+C
```

When adding a negative number to a positive number, switch the two numbers around and change the addition to subtraction. This allows you to get rid of the negative sign.

```
: - A+B→C  
can be  
: B - A→C
```

You can often times rewrite math expressions using the built-in keys and characters. When you have a number that has two or more zeros, it may be smaller to write it using the little E character (which is designed for scientific notation). This character will multiply the number on its left (1 if no number is given) times 10 to the number given on the right.

```
: 50000  
can be  
: 5E4
```

If you want to use a variable to set the exponent of a number, you would have to use 10^X because the calculator doesn't allow eX . This can be replaced with the 10^{\wedge} key. This also applies to the e^{\wedge} key, the 2 key, and the 3 character.

```
:10^A+e^2-5^2+9^3  
can be  
:10^(A)-5^2+9^3+e^(2)
```

If you have a fraction that has one as the numerator, you can replace it with multiplying the denominator by the $^{-1}$ key.

```
:1/16  
can be  
:16-1
```

When you have a fraction that has an expression in the numerator that has parentheses around it and a variable in the denominator, you can sometimes eliminate the fraction by multiplying the variable by the $^{-1}$ key and multiplying it by the expression from the numerator.

```
:If (A+B)/C  
can be  
:If C-1(A+B)
```

If you raise a variable or value to some fractional power with one in the numerator, you can just take the denominator of the fractional power and then multiply it by the $xroot$ character and the variable or value.

```
:A^(1/B  
can be  
:BxA
```

Always do all the operations you can ahead of time. This eliminates some of the operations that the calculator has to do.

```
:33+A(8/2→B  
can be  
:33+4A→B
```

Write and calculate expressions in one step instead of several steps.

```
:2BC→D  
:3A→E  
:D+E→F  
can be  
:2BC+3A→F
```

One of the basic math rules is that multiplying one times any variable is equal to the variable. So, you don't need to put the one in front of the variable.

: $1A + 3 \rightarrow B$
can be
: $A + 3 \rightarrow B$

When adding two variables of the same type together, you should add up the number of times the variable appears and multiply that value by the variable.

: $A + 3A \rightarrow B$
can be
: $4A \rightarrow B$

Rewriting division with multiplication is useful when multiplying is smaller. You take the denominator and then change it to the equivalent for multiplication.

: $(X+1)/10$
can be
: $.1(X+1)$

The distributive identity should be used when you have three or more variables that share a common number or variable. You take that common number or variable out and distribute it to all of the variables.

: $CA + CB + C^2 \rightarrow D$
can be
: $C(A + B + C \rightarrow D)$

The multiplicative inverse identity is used when you have an expression where the same variable or value is in the numerator and denominator. You can remove the variable or value because it is canceled out.

: $2A / (2BA)$
can be
: $1/B$

When you have a fraction that has a fraction as its denominator, you can sometimes use the division inverse identity. If the numerator of the first fraction is one, you can flip the second fraction causing the first fraction to disappear.

: $1 / (4/A)$
can be
: $A/4$

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/optimize-math>

Optimization: Putting Ans Into Practice

The Ans variable (last answer) is a temporary variable that can hold any variable. Ans is changed when there is an expression or variable storage or when pausing with the Pause command. It is mostly useful when you are just manipulating one variable. To use Ans just put an expression on a line by itself; it will automatically be stored to Ans. You can then change the expressions on the next line where the variable was called and put Ans there instead.

```
:getKey→A  
:B+(A=26)-(A=24→B  
can be  
:getKey  
:B+(Ans=26)-(Ans=24→B
```

If you have more than one line that calls the variable, you should just keep the variable. However, for the first line that calls the variable you should change the variable to Ans.

```
:getKey→A  
:B+(A=26)-(A=24→B  
:C+(A=34)-(A=25→C  
can be  
:getKey→A  
:B+(Ans=26)-(Ans=24→B  
:C+(A=34)-(A=25→C
```

If you store the same value to two or more variables one after the other, use Ans for each one after the first variable.

```
:500→A  
:A→B  
:A→C  
can be  
:500→A  
:Ans→B  
:Ans→C
```

When there is a common expression that is on multiple lines, it is sometimes smaller to put the expression on its own line and then change the expression on the other lines to Ans.

```
:30+5A→B  
:Disp 25A  
:Disp 30+5A  
can be  
:30+5A→B  
:Disp 25A  
:Disp Ans
```

When you use the same text many times in close proximity, you should put that text on its own line and replace it with Ans wherever it occurs.

```
:Disp "Hello  
:Disp "Hello  
:Disp "Hello  
can be  
:"Hello  
:Disp Ans,Ans,Ans
```

For complex calculations, there are often multiple parts that are the same. You should take out the most common part and put it on its own line. If there are several common parts, you should take out the part that will result in the greatest size reduction. You then replace that part, wherever it occurs, with Ans.

```
:2A/(BC)+(BC)2→A  
can be  
:BC  
:2A/Ans+Ans2→A
```

When dealing with text there are often situations where the same text is repeated multiple times. Rather than writing out the long string of text, it is sometimes possible to rewrite it using Ans. Put the common part of the text on its own line and on the next line concatenate (add together) with Ans however many times is needed to make the string.

```
:" →Str1 //20 spaces  
can be  
:" //5 spaces  
Ans+Ans+Ans+Ans→Str1
```

If you use the sub(command to get the appropriate part of some text based on certain conditions, you can sometimes get rid of the sub(command and just use Ans. You would put each piece of text on its own line, and then put the condition before it.

```
:Input sub("GiveTake",1+4(A=1),4)+" candy?",Str1  
can be  
:"Give  
If A=1:"Take  
Input Ans+" candy?", Str1
```

With Repeat loops, you can sometimes put Ans in the condition instead of the variable. Even if Ans were 0 at the beginning of the loop, the code will work, since a Repeat loop will always cycle once before the condition is checked.

```
:Repeat A  
:getKey→A  
:End  
can be  
:Repeat Ans
```

```
:getKey→A  
:End
```

When the condition in a Repeat loop has a common part that is repeated multiple times, you should put the common part at the end of the loop and replace the common part in the condition with Ans.

```
:Repeat A=2 and B=1 or A=2 and B=3  
:getKey  
:A+(Ans=26) - (Ans=24→A  
:End  
can be  
:Repeat Ans and B=1 or Ans and B=3  
:getKey  
:A+(Ans=26) - (Ans=24→A  
:A=2  
:End
```

Many times in If-Then-Else conditionals the same expression or string of text appears in both the true and false parts. You should put this expression or string of text before the If-Then-Else conditional and then replace it in the conditional with Ans.

```
:If B  
:Then  
:Disp "Hello  
:2Bnot(A→C  
:Else  
:Disp "Hello  
:3→D  
:End  
can be  
:"Hello  
:If B  
:Then  
:Disp Ans  
:2Bnot(A→C  
:Else  
:Disp Ans  
:3→D  
:End
```

When you have two or more strings of text that share a common part, you should take that common part out. You then can replace it with Ans and concatenate Ans to the strings.

```
:Disp "Hello World  
:Disp "Goodbye World  
can be  
:"World  
:Disp "Hello "+Ans  
:Disp "Goodbye "+Ans
```

When you have two If conditionals that have math opposite conditions and they display text, it is

sometimes possible to remove one of the conditionals and use Ans. Take the text from the first condition and put it on its own line. Then put the second conditional and the text on the next line. You then put the display Ans on the last line.

```
:If A≤B  
:Disp "Higher  
:If A>B  
:Disp "Lower  
can be  
:"High  
:If A>B  
:"Low  
:If A≠B  
:Disp Ans+"er
```

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/optimize-ans>

Optimizations: Displaying Text

If you have a string of numbers that you are displaying, you don't need to put quotes around the numbers. You should only use quotes if you want to keep any leading zeros.

```
:Disp "2345  
can be  
:Disp 2345
```

Use the Disp command instead of the Output command when displaying text on the first line of the homescreen. You can just add spaces to the text to move it to the correct location.

```
:Output(1,2,"Hello  
can be  
:Disp " Hello
```

When displaying the same text or variable on three or more lines, use a For loop. A For loop can also be used when the display is changing by a constant increment.

```
:Output(3,3,1  
:Output(4,3,2  
:Output(5,3,3  
can be  
:For(X,3,5  
:Output(X,3,X-2  
:End
```

When the text in an Output command is more than sixteen characters, it will wrap around to the next line. When you have two or more Output commands that display text on different lines, you can sometimes put the text together and add blank spaces between it to make it go to the next line in the desired location.

```
:Output(1,6,"Hello World  
:Output(2,2,"Version 1.0  
can be  
:Output(1,6,"Hello World Version 1.0
```

Using the Disp command, you can display text and variables at the same time by putting a comma between each one. Because this can hinder readability, this should only be done when just displaying variables.

```
:Disp A  
:Disp B  
can be  
:Disp A,B
```

When you have a list of Disp commands that you pause, you can take the text or variable from the last Disp command and place it after the Pause command as its optional argument, allowing you to remove the last Disp command.

```
:Disp "A=  
:Disp A  
:Pause  
can be  
:Disp "A=  
:Pause A
```

You can often remove Disp commands by building a string of text (putting the addition operator between each part of the text) and then displaying the text with one Disp command. This can be useful when you have two conditionals that are opposites that display text.

```
:If A≥10  
:Disp "Hello  
:If A<10  
:Disp "Goodbye  
can be  
:"Hello  
:If A<10  
:"Goodbye  
:Disp Ans
```

When you have two or more Disp statements inside an If-Then conditional, you should combine the Disp statements so you can change the If-Then conditional to an If conditional.

```
:If A>B  
:Then  
:Disp "A is greater  
:Disp "than B  
:End  
can be  
:If A>B
```

```
:Disp "A is greater","than B
```

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/optimize-text>

Optimization: Storing Variables

Although it is common to initialize variables for planned use, you should avoid initializing variables that you don't need or that are initialized further down in the program. The reason is because storing to variables really slows a program down (especially inside loops) and there is no point in initializing a variable twice.

```
:2→A  
:If B  
:Then  
:2→A  
:Else  
:-2→A  
:End  
can be  
:If B  
:Then  
:2→A  
:Else  
:-2→A  
End
```

When a number is used many times in a program, you should store it to a variable and then just call the variable instead of writing it out every time. This also applies to text that should be put in a string.

```
:Disp "Hello  
:Disp "Hello  
:Disp "Hello  
can be  
:"Hello→Str1  
Disp Str1,Str1,Str1
```

You can also put common variables or expressions in a string variable, and then use the `expr` command to reference them. This can be used in conjunction with other variable commands. This also gives you more variables to use.

```
:Disp 5int(B/7  
:Disp 5int(B/7  
can be  
"5int(B/7→Str1  
:Disp expr(Str1  
:Disp expr(Str1
```

You should reuse variables that have no specific function or that don't need to be saved.

```
:For(X,1,100
:End
:For(Y,1,50
:End
can be
:For(X,1,100
:End
:For(X,1,50
:End
```

When storing the same large number in two or more variables, you should store the large number in the first variable and then store the first variable into the rest of the variables.

```
:7112→A
:7112→B
:7112→C
can be
:7112→A
:A→B
:A→C
```

When calculating several repetitive trigonometric or other math functions in a program, it is sometimes faster to just store the values in a list and recall the values when needed.

```
:For(A,0,10
:Text(6A+1,1,10cos(A)
:End
can be
:For(A,0,10
:10cos(A→L(A
:End
:For(A,0,10
:Text(6A+1,1,L1(A
:End
```

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/optimize-variables>

Optimization: The Graph Screen

Although the screen is 95 pixels wide and 63 pixels tall, the bottom row and far right column of pixels are unusable. So, most people set the graphscreen dimensions to 94 and 62. This itself should be replaced with storing 1 into deltaX and deltaY.

```
:0→Xmin:94→Xmax  
:0→Ymin:62→Ymax  
can be  
:0→Xmin:1→△X  
:0→Ymin:1→△Y
```

The Text command can display both variables and text at the same time on the same line. This allows you to sometimes remove multiple Text commands and just use the first one.

```
:Text(5,5,A  
:Text(5,9,"/  
:Text(5,13,B  
can be  
:Text(5,5,A,"/",B
```

The Pxl-On command is faster than Pt-On, and it should be used whenever possible. Pt-On is also affected by the screen dimensions, while Pxl-On is not.

```
:Pt-On(5,5  
can be  
:Pxl-On(5,5
```

The Line command has an optional fifth argument that controls whether the line will be drawn (the argument should be one) or erased (the argument should be zero). The default is one, and it should be left off when possible.

```
:Line(5,5,10,5,1  
can be  
:Line(5,5,10,5
```

When turning multiple pixels in a straight line on or off, use a For loop instead of using the individual pixel commands.

```
:Pxl-On(5,5  
:Pxl-On(5,6  
:Pxl-On(5,7  
:Pxl-On(5,8  
can be  
:For(X,5,8  
:Pxl-On(5,X  
:End
```

When you are changing the same pixels from on to off or vice versa in a loop, use the Pxl-Change command instead of the individual pixel commands.

```
:For(X,5,8  
:Pxl-On(5,X  
:End  
:For(X,5,8
```

```
:Pxl-Off(5,X  
:End  
can be  
:For(X,5,8  
:Pxl-Change(5,X  
:End
```

When you have multiple pixels in a straight line that you turn on or off, you can sometimes replace the Pxl-On commands with Line commands.

```
:Pxl-On(5,5  
:Pxl-On(5,6  
:Pxl-On(5,7  
:Pxl-On(5,8  
can be  
:Line(5,5,5,8
```

The Pt-On and Pt-Off commands have an optional third argument that should never be used when one is desired because one is the default.

```
:Pt-On(5,5,1  
can be  
:Pt-On(5,5
```

The optional third argument for Pt-On and Pt-Off should be used when you want to turn on or off a 3x3 outline of a box (the argument should be two) or a 3x3 cross (the argument should be three). This can be used instead of the individual commands.

```
:Pt-On(A,B-1  
:Pt-On(A,B  
:Pt-On(A,B+1  
:Pt-On(A-1,B  
:Pt-On(A+1,B  
can be  
:Pt-On(A,B,3
```

When wanting to clear large spaces of the graph screen, you should use the Line or Text commands instead of the pixel commands, when possible. Both of these commands are faster than the pixel commands.

```
:Pxl-Off(5,5  
:Pxl-Off(5,6  
:Pxl-Off(5,7  
:Pxl-Off(5,8  
can be  
:Line(5,5,5,8,0
```

The Circle(command has an alternate syntax. When a complex list such as $\{i\}$ is added as the 4th argument, "fast circle" mode will be turned on, which uses the symmetries of a circle to save on trig calculations, and draws a circle in only 30% of the time it would normally take.

Since the Circle(command is speed-challenged at best, you should always use this optimization when drawing circles.

```
:Circle(0,0,5  
can be  
:Circle(0,0,5,{i
```

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/optimize-graph>

Optimization: User Input

The Input command has an optional display message argument that can be either text or a string. This display message can be used to tell the user what type of value to enter or to show what the variable is for. The Input command should be used instead of using a Disp command in conjunction with the Prompt command.

```
:Disp "Guess  
:Prompt A  
can be  
:Input "Guess?",A
```

If you have two Input commands that have display messages that are positioned between a conditional, you can often remove one of the Input commands and then use the sub command to display the appropriate display message. This allows you to get rid of the conditional. You can also take out the common part of the display message and add it to the substring part of the display message.

```
:If A=1  
:Then  
:Input "Take candy?",Str1  
:Else  
:Input "Give candy?",Str1  
:End  
can be  
:Input sub("GiveTake",1+4(A=1),4)+" candy?",Str1
```

The Prompt command can be used with more than one variable. If you have a list of prompt commands, you should put all of the variables on the first Prompt command, separating each variable with a comma. This allows you to get rid of the rest of the Prompt commands.

```
:Prompt A  
:Prompt B  
:Prompt C  
can be  
:Prompt A,B,C
```

The Prompt command should be used instead of the Input command when you have the display message show what the variable being stored to is. And if there are multiple Input

commands, you can reduce them to just one Prompt command.

```
:Input "A=?",A  
:Input "B=?",B  
:Input "C=?",C  
can be  
:Prompt A,B,C
```

When doing calculations and user input, you should move the calculations before the user input. The delay before the user input is not important because people simply can't type fast enough to notice it.

```
:Input "NAME:",Str1  
:3B/7→L1(1  
:2A+5B→C  
can be  
:3B/7→L1(1  
:2A+5B→C  
:Input "NAME:",Str1
```

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/optimize-input>

Piecewise Expressions

Piecewise expressions are a shortcut to handling conditions in math statements. They can be used for turning an If block or several such blocks into a single line. Although the TI-83 manual recommends them for graphing piecewise functions, they are very important to programmers as well — it would not be an overstatement to say that the use of piecewise expressions revolutionized TI-Basic when it was introduced.

The Concept

The general form of a piecewise expression is $(\text{expr } \#1)(\text{condition } \#1) + (\text{expr } \#2)(\text{condition } \#2) + \dots \rightarrow \text{result}$. Usually, condition #1, condition #2, and any other conditions are mutually exclusive — only one of them can be true at a time. In this case, the piecewise expression can be interpreted as follows:

- if condition #1 is true, return the value of expr #1
- if condition #2 is true, return the value of expr #2
- ...
- if condition #X is true, return the value of expr #X

A classic example of a piecewise function is absolute value, which strips a number of its sign. Forget for a moment that the abs(command exists, and picture code that would do its job. A possible solution relies on the If command:

```
:If A≥0  
:Then  
:A→B
```

```
:Else  
:-A→B  
:End
```

Using piecewise expressions, we can write this as:

```
: (A) (A≥0)+( -A) (A<0) →B
```

Most of the parentheses are unnecessary, only here for clarity. If you're comfortable with piecewise expressions, you can strip the extra parentheses to get this version:

```
: A(A≥0)-A(A<0 →B
```

Why does this work?

Believe it or not, the calculator does not make special cases for piecewise expressions. Instead, this technique relies on the convention known as Boolean logic. According to Boolean logic the number 1 represents "true" in logical expressions on the TI-83, while 0 represents "false".

In the case of a properly written piecewise expression, only one of the conditions will be true, and the rest will be false. That condition's expression will be multiplied by 1, and the others by 0. When the results are added, this gets rid of the unwanted expressions, leaving only the one with a true condition.

Optimization

Now that we know how this technique works, we can optimize such expressions while keeping the result the same. For example, here is part of the code for moving a cursor on the screen, as a piecewise expression:

```
:getKey→K  
:(X-1)(Ans=24)+(X+1)(Ans=26)+(X)(Ans≠24 and Ans≠26) →X  
:(Y-1)(K=34)+(Y+1)(K=25)+(Y)(K≠34 and K≠25) →Y
```

Notice that all three pieces of the first expression contain X, and all three pieces of the second expression contain Y. In such cases, we can take out the common part of the pieces, without changing the result:

```
:getKey→K  
:X-(1)(Ans=24)+(1)(Ans=26)+(0)(Ans≠24 and Ans≠26) →X  
:Y-(1)(K=34)+(1)(K=25)+(0)(K≠34 and K≠25) →Y
```

Finally, we can cancel the unneeded parts. Many of the parentheses are unnecessary, but it's also pointless to multiply something by 1, so we can get rid of the (1) parts entirely. Finally, the parts multiplied by 0 are redundant.

```
:getKey→K  
:X-(Ans=24)+(Ans=26→X
```

```
: Y - (K=34) + (K=25→Y
```

The result is the movement code you may have seen elsewhere in the guide, in its fully optimized form!

Advantages and Disadvantages

Piecewise expressions are usually a better choice than clunky If statements to accomplish the same thing. They give the following benefits:

- The result is usually faster to compute, and takes less memory in the program.
- The expression takes less space on the screen to scroll through.
- Piecewise expressions can be used where If statements can't (for example, equations).

However, there are a few drawbacks you need to be aware of:

- Unlike an If statement, a piecewise expression will compute *all* its parts before returning the result.
- Complicated logic can make piecewise expressions very messy and hard to understand.

So one situation in which piecewise expressions should be avoided is one in which part of the expression takes a long time to compute. For example:

```
: If N=1
: Then
: irr(I,100,L1→P
: Else
: P(1+.01I→P
: End
```

In this case, a very complicated calculation is done in the case N=1 (if L1 is large enough, it may take several seconds). But if N is not 1, the calculation is very simple and will finish quickly. If you made this code a piecewise expression, the very complicated calculation would always be calculated, even if it's not going to be necessary.

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/piecewise-expressions>

Friendly Graphing Window

A friendly graphing window is some configuration of window variables that's most useful for your program — most commonly, because it makes the coordinate value of a pixel come out to a round value, such as an integer, or .1 of an integer.

Setting up a square window

The most basic type of friendly window is one in which a vertical distance in pixels is equal, coordinate-wise, to the same horizontal distance in pixels. This means that the ratio between (Xmax-Xmin) and (Ymax-Ymin) is 47:31. Such a setup has the useful effect that drawing a circle (for example, with the Circle(command) actually results in a circle, and not an ellipse.

This can be accomplished simply with the ZSquare command, which will alter the values of the

screen so that:

- the coordinate of the center remains the same
- the ratio ($X_{\text{max}} - X_{\text{min}}$) : ($Y_{\text{max}} - Y_{\text{min}}$) is 47:31 (approximately 1.516 : 1)
- the resulting window is larger rather than smaller than the original.

A common technique is to run the ZStandard command first, so that the center of the screen is at (0,0).

Setting up an integer square window

However, it's possible to take friendliness even further, by adding the condition that coordinate values of a pixel come out to round values without long decimals. This can be done in several ways:

Using the ZDecimal command

This is by far the simplest — the ZDecimal command will set X_{min} to -4.7, X_{max} to 4.7, Y_{min} to -3.1, and Y_{max} to 3.1. As you can see, this satisfies the condition for a square window. Also, the coordinates of pixels have at most one decimal place: pixels that are next to each other differ by 0.1 in the appropriate coordinate.

However, it has the drawback of still having a decimal point. Your drawing commands may look like $\text{Line}(-3.1, -1.7, 3.1, -1.7)$ — if you didn't have to have that decimal point there, you'd save a considerable amount of space. This is possible, using the following three methods:

An integer window with (0,0) in the center

These methods are basically divided over where the point (0,0) should be. Putting it in the center ensures that drawing things in the middle of the screen takes up little space; also, you can achieve symmetry easily by appropriately negating numbers. On the other hand, the negative sign (you'll be using one 3/4 of the time, and 1/4 of the time you'll need two) can be annoying.

The following code sets up an integer square window with (0,0) in the center:

```
:ZStandard  
:ZInteger
```

An integer window with (0,0) in the bottom left corner

This approach is optimal in terms of saving space on coordinates: they are all positive numbers with at most two digits. For this reason, it is the most widely used. The following code sets up such a window:

```
:ZStandard  
:104→Xmax  
:72→Ymax  
:ZInteger
```

An integer window with (0,0) in the top left corner

This approach is useful for when point and pixel commands need to be used together. Although putting (0,0) in the top left makes every Y-coordinate negative, the window has the useful property that it's very easy to go from point commands to pixel commands: the pixel (R,C) corresponds to the point (C,-R), and equivalently, the point (X,Y) corresponds to the pixel (-Y,X). It's somewhat trickier to set up, though:

```
:ZStandard  
:104→Xmax  
:-72→Ymin  
:ZInteger
```

Why friendly windows are useful

Throughout this article, we've only made glancing comments as to why you'd want to use friendly windows. Here is a more exhaustive explanation:

Graphs come out nicer

Even with a square window, graphs become more accurate, because they reflect the actual proportions of the equation being graphed. For example, try drawing a circle in the standard graphing window — you'll get some stretched out oval. Now ZSquare and try again. The result is much better, right?

With an integer square window, certain other imperfections of the calculator's graphing go away. For example, try graphing $Y=1/(X+1)$ in the standard graphing window. Pretty accurate, but instead of the asymptote there's a slightly diagonal line. That's because the asymptote doesn't end up corresponding to a pixel of the graph: one pixel, the curve is a very negative number, the next it's a very positive number, so the calculator tries to connect them.

Now ZDecimal and graph $1/(X+1)$ again. The nearly vertical line at the asymptote disappears: because the value $X=-1$ matches a pixel on the graph, so the calculator realizes that something is undefined there.

Another similar problem occurs with graphing $Y=\{-1,1\}\sqrt{9-X^2}$. In most graphing windows, this graph, which should come out to a circle, has some gaps near the sides. In an integer square window, such as with ZDecimal, the gaps disappear, and you're left with a perfect circle.

Coordinates are round numbers

This is a more programming-related application. If you happen to need to draw something on the screen, in a program, you're likely to need a lot of Line(or Pt-On(commands at fairly specific locations. On a normal window, such a "specific location" might end up being (2.553,0.645) or something equally ugly, with unpredictable results if you try to round it to the nearest nice value (since you don't know exactly the point at which pixels change).

With a friendly window (for this purpose, an integer window with (0,0) in the bottom left is the friendliest), every single pixel has a round coordinate value. If each value is no more than a 2-digit number, you can even compress all the coordinates of a line into a single number — this technique was used, for example, in Bryan Thomas' Contra game.

Point and pixel commands are compatible

In the case of an arbitrary window, it's fairly difficult to convert from a pixel coordinate to a point coordinate and back: the exact formula is $X=X_{\min}+C\Delta X$, $Y=Y_{\max}-R\Delta Y$ if (X,Y) is the point and (R,C) is the pixel (going from pixel to point is even more painful). However, there are many cases in which you do need to go back and forth — for example, if you need to draw both text and lines at a coordinate. With friendly windows, this formula becomes much simpler — such as $(X,Y)=(C,-R)$ for the last integer window we discussed.

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/friendly-window>

Animation

Animation is the rapid display of images on the screen to create an appearance of movement: it works by displaying an image and then moving it to a new location after a short delay has occurred. While there are many different things that you can do for animation (the possibilities are practically infinite; heck, there is an entire program directory at ticalc.org devoted to animations), almost every animation depends on For(loops.

A For(loop is a special kind of While loop, with all of the loop construction built-in: the variable that the loop uses, the starting value, the ending value, and the increment. This is important because you can use all of those things to dictate how many times the animation is displayed, the speed of the animation, and even the animation itself (using the For(loop variable as the coordinates or the text that is displayed).

Animation is commonly used at the beginning of a program or on loading screens to add some visual pop or pizazz, which gives a program an edge over similar programs. At the same time, this does not mean that you can't go overboard with animation; too much animation becomes annoying after a while. Selective animation — where it makes sense and complements the program — has the best impact in a program.

You also want to keep in mind the calculator that the animation is running on. If you created your animation on the TI-83+SE or TI-84+SE, then the animation probably won't display as you intended on a TI-83 or TI-83+ calculator (calculators that have a much slower processor; 6MHZ and 8MHZ respectively compared to 15MHZ for the TI-83+SE and TI-84+SE). Of course, there are a few other things that you need to consider, so you should read the portability page for more information.

Animation Examples

This is an example of moving text: the variables of the X or Y coördinate of the text are changed by the for(command.

The spaces before "looks" and after "huh?" are needed to delete the old text.

```
ClrHome
ClrDraw
AxesOff
For(A,1,20
Text(A,40,"This
End
For(B,1,20
Text(28,B+9," looks
Text(28,69-B,"cool
End
For(C,50,25,-1
```

```
Text(C,40,"huh?  
Text(C+6,40,<16 spaces>  
End
```

Running this code gives this program:



TODO: Add more examples

- Using pictures
- Drawing/Erasing text (changing position, size, letter by letter)
- Drawing/Erasing shapes (changing position, size, color)
- Drawing/Erasing lines (changing position, size, color)

One of the most common examples of animation that you see in games is wiping the graph screen (you can certainly wipe the home screen as well). This is usually done at the end of the game, after the player has lost, or as a transition from one level of the game to the next.

Wiping the screen involves using one or more Line(or Horizontal/Vertical commands, and then displaying the line from one side of the screen to the other:

```
:For(X,Xmin,Xmax,ΔX  
:Vertical X  
:End
```

As you can see, a vertical line is displayed from the left side of the screen to the right side, effectively shading the entire screen. Since it uses X_{min} , X_{max} , and ΔX , it will work on any screen.

Another common example is displaying text. This is commonly used on the titlescreen of a game to make the game stand out to the user. There are several different ways that you can display text, but some of the most common are: letter by letter, sliding it in from the screen side, overlapping each letter, and displaying the large text behind the small text.

Displaying text letter by letter involves placing the text in a string, and then displaying the respective substring based on where you are in the For(loop. More plainly stated, display each character by itself at the respective time.

The code for this is fairly simple:

```
:For(X,1,5  
:Output(1,X,sub("HELLO",X,1  
:End
```

Animation Length

The two different options for animation length are timed and infinite: timed means the animation lasts for a set amount of loop iterations, while infinite means the animation will go on indefinitely with no end (or at least until the user finally presses the ON key).

The way you go about making a timed animation is by simply using an additional For(loop enclosed around the animation. For example, if you want the animation from before to be displayed five times, you can just do:

```
:For(I,1,5  
...  
:End
```

There are actually two different ways to make an infinite animation: use a For(loop with a really large ending value (such as E5) or use an infinite While 1 or Repeat 0 loop. The infinite While or Repeat loop is the smaller of the two, but the For(loop has the advantage that it still allows the user to exit out of the animation.

Of course, the really long For(loop is not a true infinite loop, since it will eventually end at some point. For our purposes, however, it works quite well because the calculator will actually power down after a certain amount of inactivity (the TI-83+ and above have a built-in APD feature).

Adding a Delay

If you try out any of the examples that have been shown so far, one of the things you will probably notice is that they display so quickly that you can barely see them being displayed until they are almost done. This behavior is acceptable for some animations, such as where there is lot of things being animated at one time, but it can cause havoc for a lot of animations. The way you fix this problem is by adding a delay.

There are two basic ways to create a delay: use a For(loop or use the rand command. The For(loop is just an empty loop, meaning there are no commands or functions inside of it. The rand command's alternate syntax — rand(— generates a list of random numbers, which is a rather time-consuming operation. Both of these delay methods can be worked so that they create a small or large delay simply by changing the size of the For(loop and the number of random numbers generated respectively.

For an example, here is the text animation from before, where the word HELLO is displayed letter by letter on the first line on the home screen, with each of the two respective delay methods added to it:

```
:For(X,1,5  
:Output(1,X,sub("HELLO",X,1  
:For(I,1,20:End  
:End
```

```
:For(X,1,5  
:Output(1,X,sub("HELLO",X,1  
:rand(10  
:End
```

Each delay method has its own advantages and disadvantages. The For(loop has the advantages that using it still allows the user to do something during the delay, and it does not have any additional memory overhead like rand does. The rand command has the advantage that it is smaller in size than the For(loop.

The rand command does use some additional memory for storing the temporary list of random numbers in Ans, which may be undesirable. To avoid this, you simply have to use this somewhat longer line: If dim(rand(#. Despite the presence of an If statement, you don't have to

worry about the next line being skipped, since `dim(rand(#))` will always be true.

The other concern when using the `rand` command is that if the number is large enough, the program will run out of memory from trying to generate such a large list, and subsequently return a ERR:MEMORY error. What number is too large is dependent on how much free RAM is available on the calculator, so for some people it might be 100 while for others it might only be 50. So, if you are wanting to use a large delay, it might be better to go with a `For(` loop instead of a `rand` command.

Related to that concern is the issue of portability: a delay may be appropriate on your calculator, but it won't be on another calculator. For example, if you have a TI-83 and you use a delay for twenty iterations of a `For(` loop, that would be almost unnoticeable on the much speedier TI-83+SE and TI-84+SE calculators. Conversely, if you write your program on a TI-83+SE, the delay would be much longer on a TI-83 and TI-83+, to the point that the animation would slow to a crawl.

With exception to assembly libraries, there is no viable way to check what calculator a program is being run on. A good alternative is to find the appropriate delay for each calculator, and then take the average for the delay that you use. This happy medium is just a simple fix, and really all you can do is just keep the other calculators in mind when deciding how much delay to use.

Allowing User Exiting

One of the main considerations that you have to make when using animation in a program is whether the user can exit the animation at any time they want. This applies to animations of any length, but it especially applies to long animations. This is because the user has to wait until the entire animation is finished before they can move on to the rest of the program, which is extremely annoying from the user's point of view (see program usability for more information).

There are a couple different ways you can fix this problem. The first way is to add some `getKey`'s throughout the animation to check for user key presses; and if you find any, you exit the animation.

Since the animations use `For(` loops, and we want to exit out of them before they have finished, you can do this by storing something at least equal to the end value to the variable used in the `For(` loop. For example:

```
:For (C,61,32,-1  
:Px1-On(C,47  
:If getKey:32→C  
:End
```

While this approach works quite well if your animation only consists of one `For(` loop, it doesn't work when you have two or more `For(` loops that you need to exit out of. The problem is that if you exit out of the first loop early, you then need to skip the rest of the `For(` loops in the animation.

Unfortunately, there is no real easy way to go about doing this. One option is to use branching to jump out of the `For(` loops to go to a `While 0` loop internal subprogram. The reason for doing this, of course, is to avoid creating a memory leak.

Because using branching can get rather messy, another option is using an additional variable to act as a flag. You just set the variable to an off state (zero is the standard value), and then change it to an on state (achieved by inverting the flag variable's value) when the user has pressed a key.

For example, here is an animation that displays the word HELLO letter by letter, and then erases each letter starting from the "O". If the user doesn't exit the animation early, it will be played 100 times before it is finally finished. Note the first example uses the branching while the second example uses the A variable as a flag.

```
:For(I,1,E2  
:For(X,1,5  
:Output(1,X,sub("HELLO",X,1  
:If getKey:Goto A  
:rand(10  
:End  
:For(X,1,5  
:Output(1,6-X," "  
:If getKey:Goto A  
:rand(10  
:End:End  
:While 0:While 0  
:Lbl A  
:End:End
```

```
:DelVar A  
:For(I,1,E2not(A  
:For(X,1,5not(A  
:Output(1,X,sub("HELLO",X,1  
:If getKey:not(A→A  
:rand(10  
:End  
:For(X,1,5not(A  
:Output(1,6-X," "  
:If getKey:not(A→A  
:rand(10  
:End  
:End
```

Those two options should generally suffice for most animations, but a third option available is to simply rewrite the animation. There is no hard and fast way to rewrite an animation, but it generally just involves thinking about the animation and seeing if there is an alternative way of implementing it.

One common way to rewrite animations where you are moving back and forth (or displaying and erasing text) is by combining the two For(loops into one, and using some additional variables to keep track of the current direction (or if it should be displayed or erased). When an edge is reached, you then just invert the variables values from negative to positive and vice versa.

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/animation>

Custom Menus

Menus are often used in programs to choose options, allowing a program to have multiple functions, or a game to have extra features. Though the Menu(command creates a perfectly functional menu, sometimes you want your program to use something more fancy, to have a differently-functioning menu, or just to stand out from the others with a menu that looks different.

However, a custom menu is usually 2 to 3 times the size of the same menu written with the Menu(command - the difference is between a menu that takes up 150-200 bytes and one that takes up 50-100 bytes.

Basic Menus

This covers the basics of creating a functioning custom menu. Remember to set up the graph screen before displaying the menu, to avoid something like axes covering the menu.

Numerical Input

The simplest way to create a custom menu is just to display the title, and a numbered set of options. Then, wait for a number key to be pressed, and act accordingly.

The following code is an efficient way of waiting until a number key is pressed, and converting it to a number N:

```
:Repeat 2>abs(5-abs(5-abs(Ans-83)))
:getKey
:End
:round(13fPart(Ans/13))→N
```

But you might not always need to convert the number at all. If all you're going to do with the option is go to a different part of your code, you might just want to use the getKey value for comparisons. And if you only have a small number of options, it's easier to check for them all explicitly rather than using the abs(command as above: for example, Repeat max(Ans={92,93,94}) will check for the keys 1, 2, and 3.

Arrow Key Input

If you want to get slightly more fancy, you could provide a cursor, such as an arrow or ">" symbol, that points to the selected option (there are many ways to display this). The first thing you'd do is display the title and the options, just as in the previous case (except you wouldn't need to number them). Then create a variable that stores the option currently chosen.

For the rest of the code, you would need a loop, structured roughly as follows:

Loop until a selecting key (such as enter) is pressed
Display a cursor at the currently chosen option
Wait for a key to be pressed
Erase the cursor
If an arrow key was pressed, change the chosen option variable
End of the loop

It is important that you actually wait for a key to be pressed, instead of just storing the key to getKey. The loop will work both ways, but if you don't wait for the key, then it will go through the loop even when no key is pressed, erasing and redrawing the cursor, which causes flicker. Another way to eliminate the flicker is to only erase the cursor if an arrow key was pressed.

You can use Boolean optimizations to quickly adjust the option based on the arrow keys:

```
:N+(K=34 and N<(# of options))-(K=25 and N>1→N
```

Another possible optimization is to use the row coordinate of the option, rather than the option number, for the value you store (but then you need to modify it by the row difference between two options, rather than 1, when arrow keys are pressed)

Labels vs. Values

The Menu(command goes to a label when an option is selected, whereas both of these methods return a value. It's fairly easy to go back and forth between these methods. To convert from a value to going to labels, add If statements like:

```
:If N=92
:Goto 1
```

To convert the Menu(command from going to labels to returning a value, use code like this:

```
:1  
:Menu("TITLE", "OPTION 1", 1, "OPTION 2", 2, "OPTION 3", 3  
:Lbl 3:Ans+1  
:Lbl 2:Ans+1  
:Lbl 1
```

1 is stored to Ans. If 1 is chosen, the Menu(command goes to Lbl 1, and this code finishes with Ans=1. If 2 is chosen, the Menu(command goes to Lbl 2, where Ans is increased to 2, then the code goes to Lbl 1 and finishes. If 3 is chosen, the Menu(command goes to Lbl 3, which has 2 Ans+1 commands after it, so Ans is increased twice: to 3.

However, it's usually easy to avoid using labels with menus.

Advanced Menus

This section covers advanced techniques your custom menus might use.

Multi-page Menu

(this section is based on [this menu routine](#) by Steve Hartmann)

A multi-page menu could be used for as many options as you wanted, and is another reason to use a custom menu routine. To create a multi-page menu, you would need a loop which displays the current page (most likely with some If statements), then does the necessary operations for a normal menu until either an option is selected or the left/right arrow keys are pressed. If an option is selected, obviously you exit the loop; otherwise, you change the page number but stay in the loop.

The most complicated situation you could be in is a multi-page menu operated completely by arrow keys. Here, you'd use a total of three nested loops: one for the page, another for the menu itself, and a third for waiting for a key.

Selecting Options

In an arrow-key operated menu, you have several options for the cursor. The simplest is to draw some sort of symbol next to the option currently selected. This could be embellished by animating the symbol - a little tricky, because it must be done inside a getKey loop. Here is an outline of the code to do so:

```
:Repeat K  
:For(I,1,(some limit))  
:(draw Ith step of animation)  
:getKey→K  
:I+(limit)*Ans→I  
:End  
:End
```

Inside the getKey loop, a For(loop goes through all the frames of an operation. This by itself would take too much time - even if you pressed a key, you would have to wait for the animation to cycle entirely. To prevent this, we add the line I+(limit)*Ans→I. Ans holds the value of getKey, which is 0 if no key was pressed, and not zero otherwise. So if no key was pressed,

we're adding 0 to I, which does nothing. If a key was pressed, however, we add a large value to I which puts it beyond the range of the For(loop, to exit the For(loop immediately. If the For(loop has a small limit, like For(I,1,10), you can simply add getKey to I, since getKey is at least 11 if it's not 0.

Program Structure

Most people reading this page probably are familiar with the reasons to avoid Goto and labels, and do stay away from them usually. However, menus complicate the situation enough that a lot of calculator programmers give up and use labels anyway, creating a program that's impossible to understand or to maintain because of the complex web of Goto commands. This doesn't need to happen - you can structure your code so that labels are unnecessary. The important part is to create this structure first, and then build the program around it, rather than writing a program and trying to tack a menu onto it later.

Suppose your menu chooses among several options which should run and go back to the menu, and a final Quit option. The structure for your program could look like this:

```
:Repeat choice=Quit  
:(menu that sets 'choice')  
:If choice=Option 1:Then  
:(run option 1)  
:End  
...  
:If choice=Option 35:Then  
:(run option 35)  
:End  
:End (the outermost loop)
```

The weakness in this code is that the variable your choice is stored in can't be modified by any of the options, or else you risk setting it to the value of an option that's yet to be checked for (which in that case will also run before you get back to the menu). To get around this, any options that modify this variable should DelVar it at the end of their If-Then-End block, then it won't interfere with anything else (theoretically, you could use the variable Ans, and then instead of DelVar add the line :0)

Examples

Some sample programs with a simple number menu, cursor-based menu, and animated cursor-based menu:

- [NUMBER.8xp](#)
- [CURSOR.8xp](#)
- [ANIMATED.8xp](#)

You can also download all three in [one file](#).

You can find a sample multi-page custom menu program [here](#).

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/custommenus>

Saving

The most efficient and versatile way to save data when a program exits is by using a custom

list. The list can be named after your program, hold up to 999 values, and be archived for long-term storage. Below is an example of a simple saving routine that backs up the variables A,B, and C into L SAVE and archives the list. It then unarchives and restores the data from that list.

```
: {A , B , C→SAVE  
:Archive LSAVE  
  
: SetUpEditor SAVE  
: If not(dim(LSAVE  
: {0 , 0 , 0→SAVE  
: LSAVE(1→A  
: LSAVE(2→B  
: LSAVE(3→C
```

The Explanation

First, use the syntax $\{value, value, value, \dots\} \rightarrow$ SAVE to back up as many values (usually variables) as you want. If L SAVE does not exist, it will be created with those values. If it does, the previous data will be overwritten and replaced.

```
: {A , B , C→SAVE
```

To prevent losing saved data, the list is stored in Archive memory. While in Archive memory, it will not be erased due to a RAM clear and cannot be overwritten by other programs. You might consider leaving this step out, however, to preserve compatibility with the TI-83 (which doesn't have Archive memory).

```
:Archive LSAVE
```

To load the saved data, it first must be moved out of archive memory, or an ERR:ARCHIVED will result. The most obvious method would be to use Unarchive L SAVE. However, using this command will cause problems if the list does not exist. The best command to use is SetUpEditor, which was intended for use with the built-in list editor.

As a side effect of setting up a list in the editor, SetUpEditor will create the list if it does not exist, unarchive it in archive memory, or leave it alone in RAM. In other words, SetUpEditor will always result in an unarchived L SAVE in RAM, without any errors (also see the relevant section on program cleanup). SetUpEditor can also be used on multiple lists separated by a comma.

```
: SetUpEditor SAVE
```

But what happens if this is the first time we're running the program? The answer is SetUpEditor will create our list for us, but it will have a length of 0. This allows us to check if we've saved data to it before: if we have, hopefully, it will have a length of more than that (in this case, 3). So this piece of code stores a default of {0,0,0} to the list if it's just been created (of course, you can put in anything you want as the default, or do something else entirely).

```
:If not(dim(LSAVE  
:{0,0,0→SAVE
```

Lastly, the stored data values are recalled into the variables to be restored.

```
:LSAVE(1→A  
:LSAVE(2→B  
:LSAVE(3→C
```

Protecting Saved Games

It's quite a pain when you go through all that trouble to get users to follow the game through its entirety without the user changing his/her list data to give himself/herself ultimate powers. There are a few ways to protect this from happening.

Addition Method

To protect your saved lists, you can add up all the values of the list and store it to an element in the list right before the program leaves, and check it before allowing the user to reload that saved game.

Right before quitting:

```
:sum(LSAVE,1,29→LSAVE(30 // list element 30 is used to save the
```

Checking to make sure list elements add up:

```
:If sum(LSAVE)≠2LSAVE(30  
:Disp "ERROR: DATA CORRUPTED
```

Extra list elements

Another method available to your disposal is to add extra elements that do nothing (or even better, cause errors!). No code will be provided as it is easy enough to add useless (or destructive) list elements. See [program protection](#) to get more details on destructive list elements.

Dual List method

Another thing you can do to protect saved games is to use 2 lists. Both lists will contain the same data, and can be compared to for changes made by users. To add further protection mix the order up (one list the opposite of the other).

Simple Dual List code

To get both lists the same:

```
:LSAVE1→SAVE2 // make both lists the same
```

Checking to make sure both lists are the same:

```
:If not(min(LSAVE1=LSAVE2  
:Disp "ERROR: DATA CORRUPTED
```

Backwards Order

To get both lists the same and into reverse order:

```
:seq(LSAVE1(I),I,dim(LSAVE1),1,-1→SAVE2
```

Checking to make sure both lists are the same:

```
:If not(min(LSAVE1=seq(LSAVE2(I),I,dim(LSAVE2),1,-1  
:Disp "ERROR: DATA CORRUPTED // same as above, can change error
```

CoSinTan Method

Simple, just add all the list elements except for the last one, then get sin(), cos(), or tan() of it. Then just store the result into the last element.

```
:sin(sum(LSAVE,1,dim(LSAVE)-1)→LSAVE(dim(LSAVE
```

Obviously "dim(L SAVE)" should be replaced with the dimensions of your save list, to save bytes. You can also replace sin() with any trigonometric function, such as tanh(), for added protection. Also make sure to execute a Degree or Radian command, to avoid the user being suspected for corrupting data if he's only changed a mode setting...

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/saving>

Highscores

High scores typically involve saving a combination of strings (for names) and numbers (for the scores themselves) after the program is finished. The simplest high score system will only have a single score, while a complicated one might have a series of names with corresponding scores.

Managing High Scores

If there is only one high score, managing it is simple. All you have to do is check if your score is greater than the high score, and if so, change it.

```
:If S>LHIGH(1
:Then
:Disp "NEW HIGH SCORE!
:S→LHIGH(1
:End
```

Managing a table of multiple high scores and names would be more complicated. Here is an example routine for managing a high score table with 7 scores in LHIGH and 10-symbol-long names

```
:If max(S>LHIGH:Then
:Disp "NEW HIGH SCORE!
:Input "YOUR NAME?",Str1
:sub(Str1+" (9 spaces)",1,10→Str1
:1+sum(S<LHIGH
:sub(Str0,1,10Ans-9)+Str1+sub(Str0,10Ans-8,81-10Ans→Str0
:S→LHIGH(8:SortD(LHIGH
:7→dim(LHIGH
:End
```

First, we should check if our score is even good enough to be in the high scores table. We're assuming that our high score table is kept in order, because we presumably initialized it that way (which will be discussed later), and we're going to keep it that way when we're done with this routine. So all we need to do is see if the score is greater than an element in our list:

```
If max(S>LHIGH:Then
```

Next, we should input the high scorer's name. You can make this as easy or as hard as you want to, in this example I used Input for simplicity. We also pad this by appending spaces to the end then truncating the string to 10 letters, because we want it to be exactly 10 letters long.

```
Disp "NEW HIGH SCORE!
Input "YOUR NAME?",Str1
sub(Str1+" (9 spaces)",1,10→Str1
```

Now, we find the place that the high score S got in the table. This line adds up the number of scores higher than the new one, and by adding one you get the new rank.

```
1+sum(S<LHIGH
```

Now we insert Str1 into Str0 at the correct place. First we use sub(to find all the characters before the place we're sticking it in. Str1 is added onto this, and then we use sub(again to get all the characters that go after the new name.

```
sub(Str0,1,10Ans-9)+Str1+sub(Str0,10Ans-8,81-10Ans
```

We could do the same for lists, but there's an easier way. Since the list of scores is sorted, inserting an element into its correct place is the same as adding it to the end, then sorting the

list. Finally, we remove the last score that was "bumped out" of the high score table.

```
S→LHIGH(8
SortA(LHIGH
7→dim(LHIGH
```

We're done!

```
End
```

Initializing the High Scores

Being able to add scores and names into the table would be useless without a table or names to begin with, so at the start of your program you should put in a block of code to do this.

```
: SetUpEditor HIGH
: If 7≠dim(LHIGH:Then
: " (6 spaces)
: Ans+Ans
: Ans+Ans
: Ans+Ans+Ans→Str0
: 0binomcdf(6,0→HIGH
: End
```

All SetUpEditor does is initialize the list. If LHIGH doesn't exist, it will create one with dimensions of 0. If the list does exist, nothing will be changed. As an extra check, you want to make sure that the list has 7 elements in it. If the list didn't already exist or didn't have 7 elements, the next block of code will execute.

Since the list not being there is a sign of the game being played for the first time, or that somebody tampered with the high scores, you should reset Str0 as well. We need Str0 to be 70 characters long for the names, but also add a space to the beginning and end for our computations when the person is ranked first or last.

Saving High Scores

We usually use a named list to store the high scores, due to the versatility of lists, and the fact that a named list probably won't get used by a different program (for more information, see [Saving](#)).

If we just have a score to deal with, it's simple to store it: just make it the first element of the list! However, with a complicated high score table, we'll have to store the names of the high scorers as well as their scores. So we have to find a way to convert a string to a list (and back).

This is simplest if you limit the variety of characters to be used for names (for example, uppercase letters and spaces). Then, you can store all the possible characters to a string, and use inString() to convert each character into a number - an index in that string. You would do this for all the characters, and append to the high scores. The following code is split up for clarity, but it could actually be combined into one line:

```
: " ABCDEFGHIJKLMNOPQRSTUVWXYZ
: seq(inString(Ans,sub(Str0,I,1)),I,1,70
: augment(↓HIGH,Ans→↓HIGH
```

Going the other way is equally simple. Unfortunately, there is no seq() command for strings, so you have to use a For loop instead, but other than that it's similar to the above code:

```
: " // 1 space
: For(I,8,77
: Ans+sub(" ABCDEFGHIJKLMNOPQRSTUVWXYZ",↓HIGH(I),1
: End
: Ans→Str0
: 7→dim(↓HIGH
```

High Score Security

This is an optional side to high score saving. It's impossible to make high scores completely tamper-proof, since someone could just look in the source code of your program and find out how you secure your high scores. However, you can use the random number generator to stop most casual cheaters (this is just one of many methods).

To do this, we first compute some number that depends on the entirety of the high score list. The most obvious is the sum of the elements. However, to obfuscate the process a bit more, you use the sum as the random number seed and save the first random number generated to the end of your list.

```
: sum(↓HIGH→rand
: rand→↓HIGH(78
```

To check if the high scores have been tampered with, you compute the sum of all the elements, and check if the first random number generated is the same as the one you saved. If it's not, somebody changed the scores, and the best way to punish the rascal is to reset them.

```
: sum(↓HIGH,1,77→rand
: If rand=↓HIGH(78:Then
(high scores are okay)
: Else
(the cheater has done his dirty work)
: End
```

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/highscores>

Compression Techniques

Compression involves encoding data in an alternative format that has advantages over the un-encoded format. When determining whether to use compression, the main thing you should consider is its effectiveness (i.e., how much size and/or speed gain it results in). Of course, you need to decompress the data before you can use it again.

Graphing

One of the simplest ways of compressing data is by placing several related command values in a list, instead of listing out each individual command one after the other. A good example of this is when you are displaying a picture on the graph screen using several Pxl-On(commands. The Pxl-On(command has two arguments: an X and Y coordinate. After placing the coordinates in a list, we then just loop through the list with a For(loop:

```
: {10, 10, 25, 25, 50, 50, 60, 60, 35, 35  
: For(X, 1, dim(Ans), 2  
: Pxl-On(Ans(X), Ans(X+1  
: End
```

While this compression is effective, it can be improved upon. If you look at a number, it has an integer and fraction part. These two separate, but related parts can each be isolated using the iPart(and fPart(commands respectively.

Relating this back to our previous example, we should combine the two coordinates together, placing the Y coordinate as the integer and the X coordinate as the fraction. This effectively shrinks the list in half. For extracting each coordinate, you simply use the iPart(command to get the Y coordinate and multiply the fPart(command by 100 (E2) to get the X coordinate:

```
: {10.1, 25.25, 50.5, 60.6, 35.35  
: For(X, 1, dim(Ans  
: Pxl-On(iPart(Ans(X)), E2fPart(Ans(X  
: End
```

This compression technique was possible because the Pxl-On(command has two coordinates, but it would not be very effective if we were storing the Line(command's four coordinates: X1,Y1,X2,Y2. A better alternative would be to simply put all four coordinates together in the integer of the number. Probably the best example of this technique put to use is Bryan Thomas's Contra game.

The reason that this works is because a number can have up to 14 digits, so there are plenty of digits available for us to use. For extracting the respective coordinate, you need to use a combination of iPart(and fPart(, multiplying by the related power of 10. The following code draws a line for each element in the list:

```
: {15231561, 42133313, 62186251, 48604839  
: For(X, 1, dim(Ans  
: Line(iPart(Ans(X)/E6), iPart(E2fPart(Ans(X)/E6)), iPart(E2fPart(Ans(X)/E6))  
: End
```

(Note that the lines may not display correctly if you don't have the right graph screen coordinates, so you should set your calculator to a friendly graphing window to make all of the coordinates easily-compressible two-digit numbers. In this particular example, the graph screen coordinates are supposed to be X=0...94 and Y=0...62.)

Here is a more general purpose algorithm, which extracts up to 14 one-digit numbers from a list element. It stores them in a second list for clarity:

```
: {12382740182756→L1
: 1+log(Ans(1→B
: For(A,1,B
: iPart(10fPart(L1(1)10^(-A→L2(B-A+1
: End
```

Instead of typing out $1 - 0 - ^ -$, use the $10^{\wedge}()$ token (accessible by pressing 2nd then LOG). If you can handle having the list of digits in reverse order, use this code instead:

```
: For(A,1,1+log(L1(1
: iPart(10fPart(L1(1)10^(-A→L2(A
: End
```

If you are an efficient programmer, at this point you will be wondering how we can optimize this code. This is a perfect fit for operations on lists. Compare the above code to this:

```
: {12382740182756
: iPart(10fPart(Ans(1)/10^(seq(A,A,1,1+log(Ans(1
```

This can be optimized further using [binomcdf](#) to:

```
: {12382740182756
: iPart(10fPart(Ans(1)/10^cumSum(binomcdf(int(log(Ans(1))),0
```

Complex Numbers

Besides using the integer and fraction parts of a number, you can also use complex numbers. A complex number has two parts: the real part and the imaginary part. Just like how you were able to separate the integer and fraction part of a number, you can also separate the real and imaginary parts of a complex number:

```
: real(-5+8i    // Returns -5
: imag(-5+8i   // Returns 8
```

While this doesn't have much application because using the integer and fraction part of a number is generally sufficient, it can sometimes be used in place of a 2-by-n matrix; you just use a list of complex numbers, where column 1 is the real part and column 2 is the imaginary part.

Now we'll move on to a different programming situation. In games you sometimes need a switch that tells whether something is in the on or off state. It is fairly common to see beginner programmers utilize two or more variables to keep track of the switch and alternate one variable based on the other's value.

This is an ample place for not only compression but just good logical thinking. If you remember that each variable is considered a Boolean; that means the value indicates either true or false. A false value is zero while a true value is anything else. So, you just need to check to see if the value of the variable is zero:

```
:If not(F // Check if the flag variable is zero
```

Because the F variable can be either true or false, you have the switch built-in for you. Naturally you'll want to change the value of the switch from active to inactive or vice versa, either when a certain condition happens or you have gone through the game loop or whatever, and you can do that by simply using the not operator:

```
:not(F→F // Flip the value of the flag variable
```

Matrices

The most appropriate and needed place for compression is when storing lots of data, such as levels and maps. The most common variable used by people for storing data is matrices. This is because matrices are simple to use and they make sense since they are two-dimensional. However, matrices have one major disadvantage: size.

Instead of using matrices and wasting lots of precious space, the better approach is to use either lists or strings when storing your levels. Then when you want to use a level, you just convert it to a matrix and delete the matrix after you are done with it.

Compression via Lists

Here is a sample level stored as a list, with each element representing a row to be displayed on the home screen:

```
:3→dim(L1
:If L=1:Then // If level one
:4444→L1(1
:5623→L1(2
:4567→L1(3
:End
```

Using the iPart and fPart commands that we discussed previously, you can break apart each number into its own separate integer and fraction elements. This allows us to then store each number into a specific position in the matrix, looping through it with a couple For loops:

```
:{3,4→dim([B]
:For(Y,1,3
:L1(Y→Z
:For(X,1,4
:iPart(10fPart(Z/10^X→[B](Y,5-X
:End:End
```

Compression via Strings

The formula for storing a level as a string and converting it to a matrix is not much different than it was for the list:

```

:If L=1:Then // If level 1
:"444456234567→Str1
:End
:1→F
:{3,4→dim([A]
:For(A,1,length(Str1
:exp(sub(Str1,A,1→[A]((fPart(A/4)!=0)+iPart(A/4),F
:F(F<4)+(F<4)+(F=4→F
:End

```

While this probably seems like a waste to go through all of this work just to compress a level, it is very important when you have lots of levels that you want to store. In addition, the calculator only has a limited amount of memory to begin with, so you need to take advantage of every opportunity to save memory.

Single Digit Numbers

Of course, we need to compress data too. For one-digit elements, it is rather easy.

Here is L_1 , which needs to be compressed:

```
{7,0,3,4,1,6,6,2}
```

Now, again, we use `seq(`. But, we use a few different arithmetical commands:

```
sum(seq(L1(Z)10^(Z),Z,1,8
```

Remember, 10^z is [2ND] then [LOG].

You should get this:

```
266143070
```

You'll notice that the digits are in reverse. That might be a bit confusing, but when decompressing, having it in reverse makes it smaller.

So now, we decompress:

```
seq(int(10fPart(Ans/10^(Z))),Z,2,9
```

There you go. It is decompressed back into Ans . So, now we can compress single-digit data. But what about double-digits?

Double Digit Numbers

Double digits are a little more complicated, but they are also more useful because they allow up to 100 different positive integers instead of just 10.

Several methods were mentioned previously for decompressing 2-digit numbers, if you paid

attention.

So, on to decompressing! Say you had this 4-element list stored in L₁:

```
{24,47,36,42}
```

To compress it:

```
sum(seq(10^(2Z)L1(Z+1),Z,0,3
```

The answer:

```
42364724
```

The decompression:

```
seq(int(E2fPart(Ans/10^(2Z))),Z,1,4
```

References

- Bryan Thomas and his [Contra](#) game
- Arthur O'Dwyer and his [Complete TI-83 Basic Optimization Guide](#) tutorial
- Brandon Green and his [Arrays: The Amazing Data Structure](#) tutorial
- Martin Johansson and his [String Compression](#) tutorial

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/compression>

Making Maps

For many games, the gameplay consists of going through all of the maps in the game. For example, in a maze game where each map is a different maze, when you get through the first maze you go on to the second maze, and so on until you finish all of the mazes. Another common example is an RPG where the player can move their guy around on the screen, and each screen is part of a larger map.

[fold](#)

Table of Contents

[How to Store Maps](#)
[How to Display Maps](#)
 [On the Home Screen](#)
 [On the Graph Screen](#)
[Where to Store Maps](#)
 [In the Program](#)
 [In a Subprogram](#)
[Sample Tile Based Game](#)

How to Store Maps

In order to keep track of all of the different things in a map, it obviously requires that you store the map to a variable. There are three different variables that you can use to store maps, and they each have their own advantages and disadvantages:

- **Matrices** — Matrices are best used for two-dimensional data, and are easier to access and manage than both lists and strings. At the same time, matrices are the largest variable, which can be important if you are trying to keep your program as small as possible.
- **Lists** — Lists are best used for one-dimensional data, and are faster to access than both matrices and strings. Lists also have the additional advantage that you can create your own custom lists, which decreases the likelihood that they will get messed with.
- **Strings** — Strings can be adapted for basically any context, and they are smaller in size than both matrices and lists. In addition, unlike matrices and lists which have a set maximum size (99x99 and 999 respectively), strings can be as big as RAM will allow.

Generally speaking, it's best to use the most appropriate variable for the application. Going back to the maze game, for example, a matrix would probably be the preferred variable to use because a maze has a two-dimensional shape to it.

When storing a map in a variable, you have to assign numbers to represent the different things in the map: an empty space might be zero (0), a wall might be one (1), and the player might be two (2). You would then check for these numbers when determining what to do on the map or what to allow (such as movement by the player).

Here is an example of a simple 8x16 map stored in each of three different variables (note: the respective variable is all on one line, it's just split up to make it easier to read):

```
: [[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]  
[1,2,1,0,0,0,1,0,0,0,1,0,0,0,1,1]  
[1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,1]  
[1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,1]  
[1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,1]  
[1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,1]  
[1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,1]  
[1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0]  
[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1→[A]
```

```
: {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1  
,1,2,1,0,0,0,1,0,0,0,1,0,0,0,1,1  
,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,1  
,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,1  
,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,1  
,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,1  
,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,1  
,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0  
,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
```

As you can see, the string map is the smallest of the three variables because you don't have to add all of the additional characters (the braces and the commas) like you do with the matrix and list. (You can actually get around this problem by storing your maps as a string, and then converting them to a matrix or list when you need to use them.)

How to Display Maps

Once you have your map stored in one of the variables, the next thing to do is to display it on the screen. The calculator has two different screens for displaying things — the home screen and the graph screen. The home screen is generally reserved for text, while the graph screen is generally reserved for graphics.

On the Home Screen

When displaying a map on the home screen, you use the Output(command together with a For(loop. You also need to decide what you want to display for the different things in the map. The easiest option is to just display the literal values stored in the map (i.e., 0, 1, 2, 3, etc.). A better option, although it's a little more complex, is to display a character that is representative of what the value stands for.

For example, in our maze map, we had spaces (0), walls (1), and the player (2). The spaces are what the player is going to be able to move on, so they naturally should not be displayed on the screen. A wall, on the other hand, is something that the player cannot move through, so it should be displayed on the screen. A good choice for a wall character is a 1 or an uppercase X. The player is what the user is in control of, so you want it to stand out. A good choice for a player character is an S or uppercase O.

Besides deciding what character you will use for each type of thing in a map, you also need to have a check for each one when displaying the characters. The most straightforward way to do this would be to have a separate If conditional that goes with each type of character. A better way to do this, however, is to put all of the characters in a string, and use the sub(command to access the appropriate character. For example, here is how you would display the maze from before:

```
:For(Y,1,8  
:For(X,1,16  
:Output(Y,X,sub(" X0",1+[A](Y,X),1  
:End:End
```

As you can see, we decided to use an X for the walls and an O for the player. The other important thing to notice is that we used two nested For(loops to display the map. Since the maze is two-dimensional, two For(loops are needed: the first loop gets the Y-coordinates and the second loop gets the X-coordinates. Inside the second For(loop is where we access the respective (Y,X) coordinate of the matrix and display it using Output(.

Displaying a maze level stored in a list or string is very similar, but it requires you to use a simple formula to convert the respective coordinates on the screen: $X+16(Y-1)$. For the string, you also need to use the sub(command to access the individual character in the string, and the expr(command to convert it to a number.

```
:Output(Y,X,sub(" X0",1+L1(X+16(Y-1)),1  
:Output(Y,X,sub(" X0",1+expr(sub(Str1,X+16(Y-1),1)),1
```

Where this formula comes from is that each row on the home screen is 16 characters wide, and the first row you just access the X coordinate by itself (i.e., when Y is 1, Y-1=0, and subsequently $16*0=0$). If you create a similar map on the graph screen, you need to modify this formula to match the number of characters per row on the graph screen and to take into account that the graph screen coordinates start at zero.

Besides using the formula, the string can also be displayed one other way. The Output(command will wrap any text that goes over the 16 characters of a row to the next row (and likewise with that row), and subsequently you can use a single command to display the entire map across the whole screen.

Since every space is overwritten with the map, this does not require a ClrHome command to clear previously displayed characters. Unfortunately, there is no equivalent for the graph screen.

On the Graph Screen

Displaying a map on the graph screen is essentially the same as displaying a map on the home screen, except you can make the map much more detailed because the graph screen can be manipulated on a pixel level. There are several graphics commands available:

- Pxl-On(, Pxl-Off(, Pxl-Change(, pxl-Test(
- Pt-On(, Pt-Off(, Pt-Change(
- Line(, Horizontal, Vertical
- Circle(, Shade(, Text(

When displaying a particular part of the map, you can use a combination of these commands to create almost anything you want, whether it is a wall, a monster, a rock, or even a smiley face. For example, using our 8x16 maze level from before, instead of outputting the X character for the wall in the matrix, we can draw a wall using lines:

```
:For (Y, 1, 8
:For (X, 1, 16
:If 1=[A] (Y, X:Then
:Line(4X-2, 57-6Y, 4X-2, 53-6Y
:Line(4X-3, 57-6Y, 4X-3, 53-6Y
:End:End:End
```

You should note that the window dimensions need to be X=0...94 and Y=0...62 for this example to show up correctly. In fact, with exception to the Pxl- and Text(commands, all of the graphics commands are dependent upon the window dimensions, so you should always use a friendly graphing window to ensure everything shows up as you intended.

There are several other ways to create graphics, and you should check out the [graphics](#) page for more information.

Where to Store Maps

After deciding on how you will store and display your maps, you then need to determine where you will store the maps: in the program itself or in a subprogram (or subprograms). When deciding which route to go, you need to think about how many maps you plan on having. If there aren't many maps (i.e., ten or less), they should usually all be stored in the program itself.

In the Program

For storing the maps in a program, you place each map inside its own If conditional and list the maps one after another. You then check to see which map the player needs and set up the variables for that map. Each map might also have some related information that goes along with it, such as the number of coins the player has to collect or the number of lives, so you would need to use an If-Then conditional instead:

```
:If A=1:Then // Check if the player is at map 1
:{1,2,0,0,0,1,5,5,7,3,4,2,9,8,7,1→L1
:3→B:4→C
:End
```

Once you have your maps stored in their individual conditionals, the next thing to do is decide where you want to store them in the program. An obvious choice is just placing them right

at the beginning of the program. In order to do this, however, it requires that you be able to access them. This normally entails placing a label before the maps, and then using a Goto to jump to them.

An important consideration when placing maps at the beginning of a program is what values you use for the If conditional variable. While you could use something simple like one or two, those values have a high probability of being accidentally entered in by the user or being set by another unrelated program, which would cause your program to store the respective map. What works better is to use random decimals (like .193 or 1.857) or math symbols (like e or π).

In a Subprogram

If there are several maps, you might want to consider placing them in a separate subprogram. The main reason is that when the maps are stored in the program, the program has to go through all of the code before the maps to reach a particular map. Depending on the size of the program, this can make for some major slowdowns in between maps. The internal maps also slow down the main program code itself.

Related to the first reason, the second reason to consider using a separate subprogram is that changing the maps is much easier in a subprogram. Instead of having to go through the entire program, looking for the map to change, you can just focus on one map at a time. This makes the maps more manageable, and also prevents you from accidentally changing other parts of the program.

The program code for the maps basically remains the same, it's just in another program. You might notice, though, that if you have lots of maps it takes a while for program execution to go back to the main program. This happens because program execution doesn't return to the main program until after it reaches the end of the program. You can fix this problem by placing the Return command at the end of each map conditional:

```
: If C=3:Then  
: [[0,1,0][2,1,2][1,2,0→[B]  
: 3→A:3→B  
: Return // Stop program execution and return to main program  
: End
```

Now that the maps are in a separate subprogram, you need a way to access them. When you want to access a map, you set the respective variable to the value of the map that you want, and then call the subprogram from the main program using the prgm command and the subprogram name:

```
: 2→A  
: prgmGAMELVLS
```

Unfortunately, storing the maps in a subprogram does have one major disadvantage. The user now needs another program to use the main program. If somebody tries to run the program and they don't have the maps subprogram, the main program will not work properly, and will actually return an ERR:UNDEFINED error when the program tries to call the non-existent maps subprogram. Even if this isn't your fault, the result is that your program looks very sub par.

Because of this problem, doing an all or nothing map separation (i.e., all of the maps are either stored in the program or in a separate subprogram) is usually a bad idea. The better alternative is to split up the maps so that the first ten maps (or so) are stored in the program, and the rest are stored in the subprogram. The user will now at least have some built-in maps to play,

regardless of if they have the maps subprogram. The user simply won't have knowledge of the other maps available for them to play.

Sample Tile Based Game

The game stores maps as lists, outputs the list as string, has collision detection, level advancement and a loading screen (which makes loading LONGER).

Use the arrow keys to move.

Objective is to get to the "?".

Press "x" to leave a trail of "*".

Press 2nd to quit.

Any line that has a preceding // is a comment and SHOULD NOT be entered into the calculator.

```
: " ->Str1
: " ->Str2

//Lists should obviously be one line
//Level 1
: {
1,2,1,1,1,2,1,1,1,2,1,1,1,2,1,3,
1,2,1,2,1,2,1,2,1,2,1,2,1,2,1,2,
1,2,1,2,1,2,1,2,1,2,1,2,1,2,1,2,
1,2,1,2,1,2,1,2,1,2,1,2,1,2,1,2,
1,2,1,2,1,2,1,2,1,2,1,2,1,2,1,2,
1,2,1,2,1,2,1,2,1,2,1,2,1,2,1,2,
1,2,1,2,1,2,1,2,1,2,1,2,1,2,1,2,
1,2,1,2,1,2,1,2,1,2,1,2,1,2,1,2,
1,2,1,2,1,2,1,2,1,2,1,2,1,2,1,2,
1,1,1,2,1,1,1,2,1,1,1,2,1,1,1,2->L2

// Level 2 (Just like Level 1)
: {
1,4,1,1,1,4,1,1,1,4,1,1,1,4,1,3,
1,4,1,4,1,4,1,4,1,4,1,4,1,4,1,4,
1,4,1,4,1,4,1,4,1,4,1,4,1,4,1,4,
1,4,1,4,1,4,1,4,1,4,1,4,1,4,1,4,
1,4,1,4,1,4,1,4,1,4,1,4,1,4,1,4,
1,4,1,4,1,4,1,4,1,4,1,4,1,4,1,4,
1,4,1,4,1,4,1,4,1,4,1,4,1,4,1,4,
1,4,1,4,1,4,1,4,1,4,1,4,1,4,1,4,
1,1,1,4,1,1,1,4,1,1,1,4,1,1,1,4->L3

// Level 3
: {
1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
1,2,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2,
1,2,1,2,2,2,2,2,2,2,2,2,2,2,2,2,1,2,
1,2,1,2,3,1,1,1,1,1,1,1,1,1,2,1,2,
1,2,1,2,2,2,2,2,2,2,2,2,2,2,1,2,1,2,
1,2,1,1,1,1,1,1,1,1,1,1,1,1,2,1,2,
1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,1,2,
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2,->L4

//If M = 0, we draw a map
:0->M
//Level
:1->L
```


When making maps and using this program, even numbers are considered walls and odd numbers aren't. In this program when a ?(3) is hit the user advances to the next level. This could also be useful for picking up objects.

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/maps>

Movement in Maps

Movement is commonly used in programs as a way to add user interaction. It is part of user input as it relies exclusively upon the getKey command to work. You can use movement in many programs and for many different things, including moving a cursor in a menu or moving a character around the screen. It is the latter case that we are going to explain and show here.

fold

Table of Contents

- [The Code](#)
- [Homescreen](#)
- [Graphscreen](#)
- [Simultaneous Movement](#)
- [Collision detection](#)
- [References](#)

The Code

Homescreen

This is the basis for the code used in the two later examples. A explanation for why it works can be seen [here](#)

```
:4→A  
:8→B  
:Repeat K=21  
:getKey→K  
:If Ans  
:Output(A,B," // 1 space  
:min(8,max(1,A+sum(Δlist(Ans={25,34→A  
:min(16,max(1,B+sum(Δlist(K={24,26→B  
:Output(A,Ans,"X  
:End
```

Graphscreen

This is the same code as the first, but it has the graphscreen initialization process at the beginning, and you have to switch up the keypress codes.

```

:Zstandard
:104→Xmax
:72→Ymax
:Zinteger
:1→A
:1→B
:Repeat K=21
:getKey→K
:line(A,B,A,B,not(K
:min(94,max(0,A+sum(Δlist(K={24,26→A
:min(62,max(0,B+sum(Δlist(K={34,25→B
:End

```

Depending on what is being moved, the code might need to be revised. This particular code will move a pixel, or you can make it a line if you want. However, to move sprites, you will need to add to the coordinate variables instead. If you are moving a group of pixels, it would be ideal to hard code it.

Simultaneous Movement

Once you have learned how to create simple movement, the next natural step is to add some enhancement to make it more complex. One of the most common things desired is simultaneous movement — moving multiple things at the same time. Unfortunately, real simultaneous movement isn't really possible because of the limitations of the calculator, but you can emulate it.

When moving things, you need to be able to keep track of their position on the screen and the number of things. While the fastest way would be to use individual real variables for each thing, the best approach in terms of speed and size is a list and real variable respectively.

Before you initialize the list, it is good to consider how many things you want to allow on the screen at any one time. This is an important consideration because the more things you need to keep track of, the slower the program runs. A good range to shoot for is 5-15.

Here is what the code looks like so far:

```
:DelVar ADelVar L110→dim(L1
```

We are using the A real variable as the counter and the L1 list variable to keep track of the 10 object positions on the screen. We chose to initialize the list elements to 0 because that is our flag to determine if the object is active or not.

Now when you want to add another object, you simply need to increment the counter and then store the object's position on the screen to the list. You also need to remember to check that you haven't exceed the maximum number of allowed objects on the screen. You can combine the X and Y screen coordinates together into one list element using compression.

```

:A+1→A
:If A<11
:YE2+X→L1(A

```

You also need to check for when a thing goes off the screen. When this happens, you first look

at the counter to make sure it isn't at 0, and then loop through the thing positions and move all the things to the previous list element. You then decrement the counter.

```
:If A>1:Then  
:For (X,1,A-1  
:L1(X+1→L1(X  
:End  
:A-1→A  
:End
```

When moving these things, you simply loop through the positions list and then change the position of whatever thing you want. You basically are moving one thing at a time and then switching to the next thing once it is done.

Collision detection

If you want to restrict your character's movement so that it doesn't move through solid spaces such as walls, you will need some sort of collision detection. Since this example is on the home screen, the best method is to use a string. Create a string with 128 elements, leaving spaces for nothing, which will be represented as zeros for visual aid. Equal and unequal signs make good walls. Here is an example, a maze. For more info maps, go to the page [making maps](#)

```
: "=====---  
=000=000=000=0==  
=0=0=0=0=0=0=0==  
=0=0=0=0=0=0=0==  
=0=0=0=0=0=0=0==  
=0=0=0=0=0=0=0==  
=0=0=0=0=0=0=0==  
=0=000=000=000==  
=====→Str1
```

Notice how the "maze" is set up so that the outer boundaries are all walls. The advantage of this is that it allows us to save space and speed on the calculator by removing the specific boundary check. The disadvantage is that it limits the amount of characters on screen to 6x14 instead of the full 8x16.

Now we can add the collision detection code in with our original movement code. You should notice that the main difference is the player's position for movement is checked to determine if the player is going to move onto an equals sign.

Notice how there is an extra argument after the Repeat. This allows us to have the character switch to the next maze when it reaches the end. You could also use this to switch to another map at the screen's edge.

```
:ClrHome  
:4→A:8→B  
: "=====---  
=000=000=000=0==  
=0=0=0=0=0=0=0==  
=0=0=0=0=0=0=0==  
=0=0=0=0=0=0=0==  
=0=0=0=0=0=0=0==  
=0=000=000=000==
```

```

=====→Str1 //remember, 0's are spaces
:Output(1,1,Ans
:Repeat K=21 and AB=26 //AB=26 can be changed for different exit
:getKey→K
:If Ans
:Output(A,B,"_ //One space, checks for key press and erases
:sum(Δlist(Ans={25,34
:A+Ans(" "=sub(Str1,16(A-1+Ans)+B,1→A //If future coordinate is
:sum(Δlist(K={24,26
:B+Ans(" "=sub(Str1,16A-16+B+Ans,1→B
:Output(A,Ans,"X
:End
:" //second maze

```

And you can repeat this until all your mazes have run through. In addition to using strings, you can also use lists, matrices, or hardcode the whole map in if statements. The code is fundamentally the same, except there is a different formula used to display the map on the screen and you also check the available spot with that formula. Again, just try to understand the code and play around with it.

On the graph screen, you cannot make a string for collision detection. Otherwise, you would be looking at a 5985 character string! Instead, on the graph screen, you can use a command called pxl-Test(to tell you what is in the next space being moved to.

The pxl-Test(command finds the status of a pixel on the graph screen returning a 1 if the pixel is on or a 0 if the pixel is off. Therefore, if you get a 1, the character shouldn't move to the next space. If the pxl-Test(is 0, then the character moves to the next space. The following code is the base of how this works, and you can alter it to add boundary checks or advanced sprite manipulation.

```

:sum(Δlist(K={25,34
:A+Ansnot(pxl-Test(A+Ans,B→A

```

References

- Kerm Martian and his post at the UTI TI-Basic forum about keeping track of multiple shots.
- [darkstone knight's post](#) which led to the latest few updates in the formulas.

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/movement>

Custom Text Input

Custom text input is used when you want to get input on the graph screen (or the home screen, if you don't like the look of Input or Prompt). As the Input and Prompt commands only work on the home screen, the only option available is to make your own text input routine.

The Basic Routine

The core of the text input routine is using the getKey command together with a string of

acceptable characters organized to follow the order of the respective key codes, and then extracting one substring character and storing it to the outputted text string:

```
: " →Str1 // 1 space
: Repeat K=105
: Repeat Ans>40 and Ans<94 or Ans=105
: getKey→K
: End
: Ans-20-5 int(.1Ans
: If 25>length(Str1) and 0<Ans and Ans<29
: Then
: Str1+sub("ABC DEFGHIJKLMNOPQRSTUVWXYZ", Ans, 1→Str1
: Text(10, 1, Str1
: End
: End
: If 1<length(Str1
: sub(Str1, 2, length(Str1)-1→Str1 //removes initial space
```

If that sounds confusing, please let me break it down for you. We first need to initialize the string variable that we will be using to hold the text the user inputs. You can use whichever one of the ten string variables (Str0-Str9) you want. The reason we initialize the string with a single character is because the calculator returns an error if you have a null string.

```
: " →Str1
```

We now begin the main program loop. The program will loop until the user presses ENTER. After that, we loop until the user presses a letter key (as all letters are assigned key codes from 41 to 93) or ENTER.

```
: Repeat K=105
: Repeat Ans>40 and Ans<94 or Ans=105
: getKey→K
: End
```

After the user has pressed one of the necessary keys, we then need to take the respective action in the program. If the user pressed one of the letters of the alphabet, we first check to see that the string is not already at the end of the screen (i.e. its length is 25). If it is less than 25, we add that text to our string variable and display the whole string:

```
: Ans-20-5 int(.1Ans
: If 25>length(Str1) and 0<Ans and Ans<29
: Then
: Str1+sub("ABC DEFGHIJKLMNOPQRSTUVWXYZ", Ans, 1→Str1
: Text(10, 1, Str1
: End
```

We finally close the main program loop. This text input routine just has basic functionality, as it was designed to show you how to do custom text input. It's up to you whether you want to extend it to include a larger range of acceptable characters, word wrapping, or whatever feature you want to include.

Tweaking the Routine

These are advanced features for a custom text input routine.

Backspace functionality

The above routine is very limited: once we type something in, we can't go back and change it. This add-on allows the user to press the DEL (delete) key to delete the last letter typed. To add this functionality, change the first loop code from ":Repeat Ans>40 and Ans<94 or **Ans=105**" to ":Repeat Ans>40 and Ans<94 or **max(Ans={105,23}**", and add the following code right before the final End that terminates the outer loop.

If the user pressed the DEL key, we first check that the string variable has at least one character already in it (so an error isn't returned), and then remove the last character at the end of the string and redisplay the string (erasing the three spaces to the right of the last character left behind from the deleted character):

```
:If K=23 and 1<length(Str1 //if DEL pressed and some letters have
:Then
:sub(Str1,1,length(Str1)-1→Str1
:Text(10,1,Str1," //three spaces after the "
:End
```

Flashing Cursor

A flashing cursor makes it clear that you mean business, or that you mean for the user to type in text. The code for a flashing cursor should replace the ':getKey→K' in the basic routine. You must also replace 'Ans' in ':Repeat Ans>40 and Ans<94 or max(Ans={105,23}' with 'K'. 'K' must also be added onto a new line after the second 'End' after the code below. This is because the new variable 'I' messes with 'Ans'.

We start the routine normally: repeat until a key is pressed. The two Text(statements will draw a [then erase its two tails, effectively drawing a horizontal bar. The For(loop creates an artificial delay between drawing and erasing the cursor. However, we want to end the loop if a key is pressed (so we don't have to wait until the cursor finishes flashing to type in a key). That's what the I+5Ans→I statement does: if K isn't 0, it will make I greater than 30, which will end the loop. We want to erase the cursor if a key was pressed or if I=16 (halfway through the delay loop). Finally, we end both loops.

```
:Text(10,4length(Str1)-2,[ 
:Text(10,4length(Str1)-1," //1 space after the quote
:For(I,1,30
:getKey→K
:I+5Ans→I //if K isn't 0, I will go out of bounds, ending the loop
:If I=16 or K
:Text(10,4length(Str1)-2," //1 space after the quote
:End
```

Adding Number Functionality

Although this routine differs from the one above, it accomplishes the same thing. Again, thanks to DarkerLine for the keypress to letter formula. Harrierfalcon came up with the formula to convert keypresses to numbers.

```
:ClrDraw
:DelVar A15→B
:Text(1,82,"")
:Text(0,82," LET"
:" →Str1
:Repeat max(M={45,105,21
:-5→C
:Repeat M=23 or max(Ans={21,45,105}) or (not(A)M>40 and not(A)M<95
:C+1-10(C>4→C
:Text(29,B,sub(" [",1+(Ans>0),1
:Text(29,B+1,""
:If M=31
:Then
:not(A→A
:Text(0,83,sub("LETPNUM",3A+1,3
:End
:getKey→M
:End
:If min(M≠{21,105,45,23
:Then
:If A
:Then
:sub("0123456789",27-3int(.1M)+10fPart(.1M)+2(M=102),1
:Text(29,B,Ans
:Str1+Ans→Str1
:B+4→B
:Else
:sub("ABC DEF GHIJKLMNOPQRSTUVWXYZΩ",M-5int(.1M)-20,1
:Text(29,B,Ans
:Str1+Ans→Str1
:B+4→B
:End
:Else
:If M=23 and 1<length(Str1
:Then
:B-4→B
:Text(29,Ans,"
:sub(Str1,1,length(Str1)-1→Str1
:End
:End
:End
```

A=1 if NumLock is enabled, and A=0 if AlphaLock is enabled. Allow me to break it down.

```
:ClrDraw
:DelVar A15→B
:Text(1,82,"")
:Text(0,82," LET"
:" →Str1
```

I'll let you guess on this one.

```
:Repeat max(M={45,105,21  
:-5→C
```

This resets the cursor counter, and initializes the loop that won't quit until [ENTER],[2ND], or [CLEAR] is hit.

```
:Repeat max(Ans={23,21,45,105}) or (not(A)M>40 and not(A)M<95 and
```



This initializes the loop which won't quit until proper keys other than [ENTER], [2ND], [CLEAR], or [DEL] is hit.

The second boolean value is structured so that if A=0, it means AlphaLock is enabled, and if a letter key is pressed, then it will exit the loop. If A=1, then it won't work, because not(A)M will equate to 0 every time. If [VARS] is pressed, nothing happens, because there is no letter there. The second Boolean value was created in a fashion that tells if A=1, then M is left alone. If it is not, then it doesn't count.

```
:C+1-10(C>4→C  
:Text(29,B,sub(" ",1+(Ans>0),1  
:Text(29,B+1,"  
:If M=31  
:Then  
:not(A→A  
:Text(0,83,sub("LETPNUM",3A+1,3  
:End  
:getKey→M  
:End
```

The first line increments C, and returns it to -5 if C=5.

The second line outputs [or a space, if C<= 0 or C>0, respectively.

The third line clears the bracket's tails, or does nothing.

Lines 4-8 toggles AlphaLock and NumLock, and updates the display.

```
:If min(M≠{21,105,45,23  
:Then  
:If A  
:Then  
:sub("0123456789",27-3int(.1M)+10fPart(.1M)+2(M=102),1  
:Text(29,B,Ans  
:Str1+Ans→Str1  
:B+4→B  
:Else
```

This If-Then is executed only if [2ND],[DEL],[ENTER], and [CLEAR] were NOT pressed. This prevents garbled numbers with would cause ERR:DOMAIN. This is executed only if NumLock was on.

```
:Else
```

```

:sub("ABC DEFGHIJKLMNOPQRSTUVWXYZθ", M-5int(.1M)-20, 1
:Text(29, B, Ans
:Str1+Ans→Str1
:B+4→B
:End

```

This is executed if NumLock was NOT on, i.e. if AlphaLock was on.
The rest shouldn't have to be explained. Optimizations are out there...can you find them?

Advanced Editing Functionality

This routine adds several features that are not in the other versions. It allows for uppercase and lowercase text with switching in between them. It allows the user to move the cursor throughout the text and insert text and delete text where ever they would like. The user can also clear the current text and start over. Unfortunately, the code is a little hard to decipher, but we'll work through it. The main deficit of this program is that it is incompatible with the TI-83. This can be remedied, as explained below.

```

:"ABC abc DEFGHdefghIJKLMNOPQRSTUVWXYZ(Theta)!xyz
:" →Str1 // 2 spaces in quotes
:DelVar M1→P
:Repeat K=105 and 2<length(Str1
:Text(0,0,sub(Str1,1,length(Str1)-P)+"|"+sub(Str1,1,length(Str1)-P
:Repeat Ans
:getKey→K:End
:P-(K=26 and Z>1)+(K=24 and Z<length(Str1)-1→P
:M xor K=31→M
:If K>40 and K<105 and K≠44 and K≠45
:sub(Str1,1,length(Str1)-P)+sub(Str0,K-40+5M,1)+sub(Str1,1,length(
:If K=23 and P<length(Str1)-1
:sub(Str1,1,length(Str1)-P-1)+sub(Str1,length(Str1)-P+1,P→Str1
:If K=45:Then
:" →Str1 // 2 spaces in quotes
:1→P
:End
:End
:sub(Str1,2,length(Str1)-2→Str1

```



The first three lines take care of variable initialization. M stores whether capitals are enabled or not. M = 0 if its uppercase, M = 1 lowercase. Str0 contains all the uppercase and lowercase letters, including information for the last row of keys on the calculator. Str1 is the string that the text will be stored in. The information will be between the two space characters so that we can insert information at the beginning and end without having problems. P is the number of places before the end of the string to place the cursor. This is the most crucial variable, this will allow for the painless insertion of text and deleting of text anywhere in the string.

TI-83 Compatibility: Change Str0 to be the string listed above for the basic routine ("ABC DEFGHIJ...."). This, along with the other change listed below, will make this TI-83 compatible.

```
:Repeat K=105 and 2<length(Str1
```

This repeats until [ENTER] is pressed and makes sure that text has been entered into the string

before exiting.

```
:Text(0,0,sub(Str1,1,length(Str1)-P)+" | "+sub(Str1,1,length(Str1)-P
```

This displays the text at 0,0 on the graph screen. This is a difficult line so let's break down what is being outputted onto the screen.

```
sub(Str1,1,length(Str1)-P)+" | "+
```

This returns all the characters that are before the cursor and then outputs the cursor

```
sub(Str1,1,length(Str1)-P+1,P)+"           // 5 spaces after quote
```

This returns all the characters that are after the cursor and then outputs some spaces. The spaces are there because when delete is pressed, the string will get smaller, and some characters will be left over. This is to cover those characters up.

The next two lines just get the key press and store it into the variable K.

```
:P-(K=26 and Z>1)+(K=24 and Z<length(Str1)-1→P  
:M xor K=31→M
```

The first of these two lines control the movement of the cursor (stored in P). If the right arrow is pressed then P will decrement, less spaces from the end of the string. If the left arrow is pressed then P will increment, more spaces from the end of the string. The other conditions make sure that the cursor doesn't go too far in either direction.

The second of these lines controls the current capitalization state. By using the xor command, M will switch back and forth between 1 and 0 (lowercase and uppercase, respectively) every time the [ALPHA] key is pressed.

```
:If K>40 and K<105 and K≠44 and K≠45  
:sub(Str1,1,length(Str1)-P)+sub(Str0,K-40+5M,1)+sub(Str1,1,length(
```

These lines test to see if the key press is a letter key and then inserts the letter into the string. The second line concatenates three strings together. The first is all the characters before the cursor. The second is the letter that was pressed (found in Str0). The third is all the characters after the cursor. This is all stored back into Str1.

TI-83 Compatibility: In order to make this compatible with the TI-83, change the middle string that is concatenated to sub(Str0,K-20-5int(.1K),1). This, along with the other change, listed above will make this TI-83 compatible.

```
:If K=23 and P<length(Str1)-1  
:sub(Str1,1,length(Str1)-P-1)+sub(Str1,length(Str1)-P+1,P→Str1
```

These lines test to see if [DEL] was pressed and then deletes a character from the string. Since the character deleted is from behind the location of the cursor, the conditional tests to see if the cursor is at the beginning of the text (where there is no character before the cursor). The second line concatenates two strings together. The first is all the characters before the cursor

minus the last one (the character being deleted). The second is all the characters after the cursor. This is all stored back into Str1.

```
:If K=45:Then  
:" →Str1 // 2 spaces in quotes  
:1→P  
:End
```

These lines test to see whether [CLEAR] was pressed and then clears the text accordingly. Both Str1 and the location of the cursor are reset to their original values. After this line, the main loop ends.

```
:sub(Str1,2,length(Str1)-2→Str1
```

This line removes the extra space on each end of the string and returns Str1.

If you want to let the user view the current capitalization state, add the following line before the getKey loop.

```
:Text(55,90,sub("Aa",M+1,1
```

The coordinates can obviously be changed to place this anywhere around the graph screen.

References

- DarkerLine came up with the formula for translating the letter keys into the short string.
- Harrierfalcon created the formula to convert number keypresses into a short string.
- Zaphod Beeblebrox created the advanced editing functionality routine.

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/custominput>

Graphics

In building the graphics for your game, you will most likely need to use many different techniques to build the game screen, make the player, animate actions, display stats, and so on. And most of the time you will have to blend certain techniques with others specifically for the game you are producing. The main idea here is that these graphic commands and techniques are not made to stand alone. They may need to be combined, blended, isolated, and/or mixed.

fold

Table of Contents

[Text Sprites](#)

[Example](#)

[Example Explanation](#)

[Advantages and Disadvantages](#)

[Using Picture Variables](#)

[Hard Coded Sprites](#)

[Stat Sprites](#)

[Advantages and Disadvantages](#)

[Greyscale](#)

[Caching](#)

Text Sprites

Text sprites are a rarely used method of displaying sprites. This is partially due to their slowness, but they are apt for creating 2D puzzle games.

As their name implies, text sprites are sprites made of text. They generally involve using a For loop to loop through a string, and displaying each character one more space to the right than the previous. As a result, only the first column of each character is shown, allowing almost every conceivable 5*5 sprite to be shown.

Example

Consider this:

```
: "([X[(" →Str1           //the characters plus 2 spaces
: ClrDraw
: For(X,1,7
: Pause
: Text(0,X,sub(Str1,X,1
: End
```

Example Explanation

If you run this code, you will slowly see a donut being drawn out, frame by frame. Each frame introduces a new character, and as is apparent, eventually forms the sprite. And because each character is shown one pixel to the right of the previous, that character overwrites the underlying character.

You might be wondering by now why the spaces are needed. Try taking them out of the code, and see what happens. The donut becomes a sideways devil donut. Those two horns are the top and bottom pixel of the ending ")". In this case, you really only need one space, but other characters which take up more than 2 columns will need 2 or more spaces.

Advantages and Disadvantages

Text sprites are limited, as some sequences of pixels cannot be shown, like 01011 among others. Although this is very few, it still prevents certain sprites. They are also limited to 5*5, or 7*7, if you use small or big text, respectively.

Text sprites also have the annoying tendency to overwrite whatever is to the right of them. In addition, they take a few milliseconds (on an 84+SE) to appear, and thus are unsuitable for games that require speed.

Despite this, do not underestimate text sprites! Some fine examples of using them are out there, like here:

[Donut Quest II](#)

[Donut Quest II Level Editor](#)

DarkerLine made it so you can create custom tiles for Donut Quest II. Buried in the Level Editor is a program that spits out the string to display certain sprites, and also prevents you from making invalid text sprites. Invalid simply means cannot be displayed as a text sprite.

It is also worthy to note that in most cases, when using the Text() function, it draws the text and erases the pxl above the text, overall erasing 6 lines. However, on TI84's, whenever you visit the Mode menu, a flag is set that makes the small text also erase the pxl below the text. This can cause havoc in programs that use text sprites, especially because of their difficulty to track down.

The use of a simple pxl-Test() algorithm should be able to alert the user of any problems before the game starts.

There are multiple ways to remedy the Text() underline problem. The program itself can execute DispTable or the assembly program AsmPrgmFDCB058EC9. Or, you can instruct the user to visit the table (2nd TABLE), matrix editor (2nd MATRIX LEFT ENTER), or list editor (STAT ENTER) - or to "Reset Defaults" in the memory menu (2nd MEM 7 2 2).

Using Picture Variables

It is common to see people use picture variables to store a picture vital to the program, such as a title screen. This is commonly justified because drawing the picture with the individual graphics commands not only takes up lots of memory, but also takes considerably more time to display than the picture variable does (which is almost instant). However, this isn't always the best policy.

Because there are only ten picture variables (Pic0-Pic9) and they are shared by all TI-Basic programs, the picture invariably is going to get overwritten by another program. You might think that Pic9 or Pic0 aren't used very often, so they are safe to use, but that is just asking for trouble. Don't store any important picture to a picture variable.

Instead, what you should do is use the graphics commands to make the picture in the program, and then either place them in a separate program (i.e. an external subprogram) or place them at the beginning of the program (if you want to limit your use of subprograms). You can just check the first time that the program is run to see if the picture exists, and if it doesn't, create it. It's okay to use a picture variable in this scenario because you can just recreate the picture if it gets deleted or overwritten.

If you don't want to type in all the commands, a good converter program is [here](#).

Hard Coded Sprites

Hard coded sprites are those that are drawn using individual Basic Pxl-^{*}() commands. While they (typically) are faster, they also take more memory, and are less easily changed if you are making a long RPG. That means that you'll have to type out every Pxl-^{*}() command for every pixel that is on, in every tile that appears in your RPG.

An example of a hard-coded sprite looks like this:

```
:Pxl-On(X,Y  
:Pxl-On(X+1,Y+1  
:Pxl-On(X+2,Y+2  
:Pxl-On(X,Y+2  
:Pxl-On(X+2,Y
```

Fast, but takes up a lot more memory. Often times, a matrix is used to carry which pixels are turned on; this is much more effective for more tiles, since you only need the matrix, and the drawing routine handles the rest.

```
:Input "Sprite?",A
:If not(A
: [[1,0,0,0,1][0,1,0,1,0][0,0,1,0,0][0,1,0,1,0][1,0,0,0,1
:If A
: [[0,1,1,1,0][1,0,0,0,1][1,0,0,0,1][1,0,0,0,1][0,1,1,1,0
:For(A,1,3
:For(B,1,3
:If Ans(A,B
:Pxl-On(A,B
:End
:End
```

If you input 0, then an X will be drawn, but any other number makes a O. You can do this with any size sprite, provided it can fit on the calculator screen (94*62), it doesn't take up the entire memory, and you tweak the For loops to be the proper size.

Stat Sprites

Statistical sprites are created by using the Plot#(commands to draw points or lines very quickly from a pair of lists for coordinates, and are drawn whenever the graph screen is accessed. A friendly graphing window is essential to making this work. The main advantage of stat sprites is their flexibility: they can be translated (shifted), reflected (flipped), rotated (turned), and dilated (stretched) by way of simple arithmetic. This offers plenty of ease and control over the sprite display.

To handle these transformations, it is practical to keep two lists of coordinates and then use variables as modifiers on these lists. For instance, when you reflect the image horizontally, one operation would flip the entire sprite by making all the x-values negative, and another would move it forward again to the correct location by adding the right amount to the same list. Of course, these two operations simplify to merely subtracting the list from a specified number.

Here are some formulas for these transformations, with L_1 as the *x-list* and L_2 as the *y-list*:

Transformation	Formula
Horizontal translation	$A+L_1 \rightarrow L_1$
Vertical translation	$B+L_2 \rightarrow L_2$
Reflection about the x-axis	$-L_1 \rightarrow L_1$
Reflection about the y-axis	$-L_2 \rightarrow L_2$
Rotation 90° clockwise	$L_1 \rightarrow L_3$ $-L_2 \rightarrow L_1$ $L_3 \rightarrow L_2$
Rotation 90° counterclockwise	$-L_1 \rightarrow L_3$ $L_2 \rightarrow L_1$ $L_3 \rightarrow L_2$
	$L_1 \cos(\theta) - L_2 \sin(\theta) \rightarrow L_3$

Rotation by an angle of θ	$L_1 \sin(\theta) + L_2 \cos(\theta) \rightarrow L_2$ $L_3 \rightarrow L_1$
Horizontal stretch	$AL_1 \rightarrow L_1$
Vertical stretch	$BL_2 \rightarrow L_2$

Advantages and Disadvantages

Statistical sprites are faster than the traditional sprite methods and they generally require less code, but there are still some disadvantages to using them over the other techniques. After coordinates get updated, the sprite needs to be redisplayed with the DispGraph command; unfortunately, because of the way the Plot#(commands work, this also causes the graph screen to be erased every time! This makes it almost impossible for them to be used in action games or other speed-intensive settings, but they are handy for drawings that don't need to be updated regularly, such as in a splash screen or title screen.

Greyscale

Greyscale is the use of two or more pictures to create a greyscale effect. On and Off tells what the pixel state would be on what pic

pic1	pic2	pic3	result
off	off	off	white
on	off	off	light grey
on	off	on	dark grey
on	on	on	black

The N here is the one of the finance app.

```
1→N
Repeat getKey
While N≠4
real(3,N,0,1
N+1→N
End
1→N
End
```

This code is the fastest that I could think of to make greyscale. This code uses xLIB.
"real(3,N,0,1" By using xLIB, it makes it really fast.

Caching

Caching is the use of picture variable to store the initial game screen when it is first created. For example, you would create the level using whatever technique you want, and then storepic(to one of the picture variables. The advantage of caching is that it allows you to be as destructive in your graphics as you want, because you always have a backup copy of the original.

Lets look at a real world example.

Lets say you are making a graph screen based platform game where your character can climb

up ladders. When your character moves over a ladder, it essentially covers up a section of the terrain. When you move your character again and erase the previous position, you would also erase the section of terrain (ie: the ladder) and it would be gone. Using caching, this problem would be solved by recalling the picture and restoring the terrain completely. This works because RecallPic() uses OR logic. Which means the picture will be 'laid' over the existing graph screen.

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/graphics>

Program Protection

Disclaimer: Program protection not only is rather limited in its effectiveness, but also acts as a hindrance towards maintaining the open and collaborative nature of the TI-Basic community: allowing others to study and learn from your code, and to use the techniques and concepts in their programs, increases the quality of TI-Basic programs being released.

You've just finished working on your latest, greatest program. You put in a lot of time and effort creating the program, and now you want to enjoy the fruits of your labor — showing it off to your friends at school. When your friends try the game, you get positive feedback and they tell you how much fun it is, and even ask if you could put the game on their calculators.

Now, you don't mind putting the game on your friends' calculators, but you want to make sure that no one can mess with it. Once the game is out amongst the school crowd, you know that other people will want the game so you need to come up with a way to protect your program. Fortunately, there are several ways to protect a program.

Before getting to program protection, the first thing you need to do is edit lock your program. You can edit lock a program using either one of the several downloadable assembly programs or the Graph Link software made by TI. It goes without saying that you should never give someone an editable version of your program.

Once your program is edit locked, now you can add a security function. Although there are several ways to protect a program, they each have varying degrees of complexity and success. The general rule is that the more complicated the protection is, the more difficult it will be for someone to circumvent it.

fold

Table of Contents

- [Put the Code Together](#)
- [Entering a Password](#)
 - [Entering Seeds](#)
 - [Hash Functions](#)
- [Self-Modifying Code \(SMC\)](#)
- [Causing an Error](#)
- [Storing the Protection Status](#)
 - [Trial Periods](#)
 - [Authorization Required](#)
- [Keeping subprograms un-executable](#)
- [Thoughts to Consider](#)
- [References](#)

Put the Code Together

Arguably the simplest, yet most crude program protection method is just putting all of the code in the program on one line. You might recognize this as utilizing compact style, except this time it serves as a program safeguard instead of a stylistic choice.

In order to put code together, you need to separate each command with a colon (:). The colon closes everything except a literal string, in which case the colon will actually be included as part of the string. In order to prevent this from happening, you need to close the string with a quote before adding the colon.

```
:If A=2:Then  
:Disp "Hello  
:not(B→B  
:End  
can be  
:If A=2:Then:Disp "Hello":not(B→B:End // Note the closed string
```

There is one command that doesn't need a colon following after it — DelVar — but leaving it off can cause some problems. DelVar's are typically chained together with one variable after another (i.e., DelVar ADelVar B), but the DelVar command also allows you to take the command from the next line (it doesn't matter what command it is) and put it immediately after the DelVar (i.e., DelVar ADisp "TI-Basic").

This works for the majority of commands, but there are two cases in which the command will actually be ignored: the End from an If conditional and a Lbl command. Both of these cases can cause your code to not work correctly anymore, and so you should either add the appropriate colons between the DelVar's or re-organize your code to eliminate the situation entirely.

```
:If not(X:Then:-Y→Y:DelVar ZEnd  
can be  
:If not(X:Then:DelVar Z-Y→Y:End // Note DelVar's position
```

When you put the code together it will wrap around to the next line (and keep wrapping around for however many lines are needed), which is useful because the average calculator user will not be able to read and understand the code then. More importantly, if they press a key to try to mess with the code it can have dire consequences. Specifically, if the user presses CLEAR, the whole line of code (i.e., the entire program) will be deleted.

Entering a Password

If putting all of the code together on one line seems rather complicated (and maybe not worth all that effort), a simpler program protection method is having the user type in a password at the beginning of a program. You then check the password against the stored password and allow the person to play the game if the passwords match or exit back to the home screen. You can have the password be whatever you want.

```
:"5552646472→Str1 // Store the password to a string  
:For(X,1,.5length(Str1 // Loop every two characters for a key  
:Repeat Ans
```

```
:getKey
:End
:If Ans≠expr(sub(Str1,2X-1,2) // Check if the user typed the wrong
:Stop      // Stop the program and return to the home screen
:End
```

When editing the password string, you must keep the length divisible by two because of the For(loop and the If conditional check. Besides that, this code does not allow keys 102-105 to be included in the password. That shouldn't be too big of a problem, though.

Entering Seeds

You can use pseudo random number sequences as a sort of password protection. After seeding the rand command, the results generated will be unique to the seed that was chosen. If the seed takes on the behavior of a password, then a comparison of the rand function to one of its precomputed results will act as an authentication for that password.

For instance, 5→rand followed by a single use of rand will return .727803... on all calculators, so a test can be devised as follows:

```
:Input X      // Request a number
:X→rand      // Seed the random number generator
:If rand≠.7278038625 // Check if the first random number is not equal to the seed
:Stop      // Stop the program
or
:If rand≠.7278038625
:Stop
```

Only when the user inputs the correct seed (or in the latter case, stores the correct seed to rand before running the program) will he be able to venture past this part of the code. The upside to this technique is that even if he does see the code, he won't be able to figure out what the password is just by looking at that number.

Going further with this, you can test for a result that is obtained only after a specific number of *numtrials* (i.e., uses of the rand command). After storing 7 as a seed, the third result will be .577519..., so having a test similar to the one shown above will mean that the code that follows it will only work on its third execution after the seed is stored manually — adding another layer of obscurity.

Hash Functions

While using the seq(command, the calculator can still interpret keypresses and store them to getKey. One possible way you can use this feature is to make a password function that asks the user to enter in the correct password before time expires:

```
:DelVar L1seq(getKey,X,1,200→L2
:For(A,1,dim(Ans
:L2(A
:If Ans:Ans→L1(1+dim(L1
:End
:If 5=dim(L1
```

```
:If max(L1≠{55,52,64,64,72  
:Stop  
:"Success!"
```

The main problem with using this routine is that you have to create a huge list to have enough time to input a reasonable password. This can be fixed by replacing seq(getKey,X,1,200 with something that goes a little slower:

```
:seq(getKey+0rand,X,1,100)  
:seq(getKey+0dim(rand(2)),X,1,100)  
...
```

This does lose a bit of sensitivity, but this isn't a huge problem because the routine has a lot of sensitivity to begin with. Even adding +0dim(rand(2)) left the code still sensitive enough that it recorded every keypress of me simply brushing a finger across the keyboard of my TI-83+.

Put this together with the idea that we don't want to store the password itself (because that would be fairly easy to figure out), but rather a hash of the password — a numerical equivalent value for the password. This is easier than extracting the nonzero elements of a list. For example, sum(Ans is a decent option that doesn't care about the order of the keypresses. If you want an option that does, take cumSum(Ans)not(not(Ans first — this multiplies the last keypress by 1, the next-to-last by 2, the one before that by 3, and so on.

Here is an example:

```
:ClrHome  
:Disp "Input Password  
:seq(getKey+0dim(rand(2)),I,1,50  
:If 106.322402=sum(√(cumSum(Ans)not(not(Ans  
:"Success!"
```

This example will display the message Success! if you enter the password AWESOME. Obviously, one of the main programs with using a hash function is coming up with the different hashes for the passwords, so here is a program that will assist you in making the hashes:

```
:{0→L1:0  
:Repeat Ans=105  
:If Ans  
:Ans→L1(1+dim(L1  
:getKey  
:End  
:sum(√(cumSum(L1  
:DelVar L1Ans
```

Input your password and then press ENTER to get the appropriate number to test against.

Example password: HAL
Hashed result: 29.8632681

By replacing 106.322402 in the hash password program with 29.8632681, the password will be reconfigured to HAL.

Self-Modifying Code (SMC)

Another way you can protect your program is by using self-modifying code. SMC makes your code more difficult to understand, and by placing code inside a graphing variable, you are essentially hiding it. This prevents somebody who's not very knowledgeable from figuring out what it is.

A good example of this is where you have an If conditional, and you replace part of the condition with a graphing variable:

```
:If Xnot(Yint(rand  
can be  
:"not(Yint(rand→u  
:If Xu
```

If this conditional is inside a loop, then you can modify the u variable later so that its code is something different when the If conditional is checked next time. For the average calculator user, this will make your code seem obfuscated, and they will be hesitant to mess with it.

Causing an Error

Depending on the protection used, you usually want to implement an error when it has been breached. The simplest error would be a message to the user. <error status> can be anything you want: see the methods below for when to cause an error.

```
:If <error status>  
:Pause "ERROR! UNAUTHORIZED USE DETECTED!  
:Stop
```

Unfortunately, this method allows the user to know when the error occurred and remove the error code by pressing ON when the error is displayed. A more secure method uses an error caused by the calculator that cannot be traced to specific code. The drawback of this method is that a custom error message explaining the problem cannot be displayed.

```
:If <error status>  
:Goto XX
```

This code will cause a program to display ERR:LABEL because there is no label XX. It is one of the few errors that does not have the option to go to the code causing the problem, which makes it more secure. An experienced user will most likely be able to find the problem Goto, however.

The most complicated method of causing an error is to embed pieces of code that cause problems when <error status> is true. In the examples below, problems are caused when $X \neq 20$ (replace this with whatever condition you want). Unless the user is an expert, it will be difficult for the user to fix the errors.

```
:If 21=getKey(X=20 // Clear getKey  
:L1(X=20→LSAVE // Prevent saving  
:A+X→A // Use as a replacement to A+20→A
```

```
:max({17,21,35,42,55}=seq(5Aint(B(X=20)/fPart(C
    // Screw up a complicated command
    // Extremely difficult for someone else to figure out
```

Another option is to quit the program immediately. This is most effective in a large program, where users will have to pore through hundreds if not thousands of lines of code to find the problem code. In addition, make the program jump to the default quit routine instead of a new one to confuse users even more.

```
:If L1(31)=5      // Quit condition test
:Goto XX        // Default quit label
...      // Whatever code is in between the Goto and matchingLbl
:Lbl XX
...      // Stuff to do before quitting
:Stop
```

Storing the Protection Status

The other program protection methods all require one variable in which to store the protection status (the number of times the program has run for a "trial period", whether it is protected or not, etc.). You can use any variable for this, but each has its own advantages and disadvantages: a custom list is best (but somewhat difficult to implement) and a finance variable is second best.

- **Regular variables** — Have the advantage of being readily accessible, but are not very suitable because they are frequently overwritten by other programs.
- **Finance variables** — Built-in to the calculator and are somewhat hidden, so they are unlikely to be erased. You can access these variables by going into the Finance menu. The only uses of the finance variables are the Finance Application and other programs. If another program is using the finance variable you want to use, either use a different one or change the other program. However, all the finance variables are reset to zero when the RAM is cleared.
- **Custom Lists** — Can be archived, and it is unlikely that some other program would use the same list name. However, the list is visible in the Memory Management menu, and a perceptive user may realize that it is being used for program protection and change it. To counter this, you can hide the value among other values in another list used by your program (for example, save lists).

Trial Periods

If you wanted your program to only run a certain number of times (a trial period), you will have to have a counter that counts the number of times the program has run and produce an error when the limit is reached. (See above for a discussion on which counter to use.) For this example, we will use the finance variable *n* for simplicity. Add something like this to your program:

```
:n+1→n      // Increment the counter
:If n>5      // If this is past the fifth time, free trial is over
:Goto XX      // Replace this with any of error methods explained ab
```

That's it. The above code will cause an error message to be displayed after the user has

reached the end of the trial period (used the program five times). You can change this however you want to fit your needs. Since the increment comes after the error, it will continue erring each time it is run.

Authorization Required

You can also set up your program so that only authorized/licensed users can run it. This method can be combined with the above method: Users can use the programs until their free trial is up and they have to become "authorized." To "authorize" an individual calculator, set *n* to a predetermined value.

There are two ways of doing this: either type in the value manually (and use Clear Entries afterwards to prevent the user from discovering it), or transfer *n* as part of a group containing your program.

```
:If n≠20      // If n=20, the calculator is "authorized"
:Goto XX      // If not, cause problems
    // You can replace this with any of the errors mentioned above
```

You can also use this method to lock some of your program's features in the "unauthorized" versions. For instance, in this example every user can use feature one (which is part of label 1) while only authorized users can use feature two (which is part of label 2):

```
:Menu("MAIN MENU", "FEATURE 1", 1, "FEATURE 2", 2
:Lbl 1
<feature one, available to everyone>
:Lbl 2
:If n≠20:Then
:Disp "ONLY AUTHORIZED", "USERS CAN USE", "THIS FEATURE
:Else
<feature two, restricted>
:End
```

Keeping subprograms un-executable

Say you have a large program with many subprograms, the only correct way to run the program is to run the main one. So to keep subprograms from being run outside of the main one you create a pass-on key and have the subprograms check Ans to see if it matches.

```
PROGRAM:MAIN
:randInt(1,100→Z      //make a pass-on key, keeping it new each time
(...rest of code)
:Z                      //Store the key as Ans
:prgrmSUB1             //Call the subprogram
```

```
PROGRAM:SUB1
:If Ans≠Z              //Check the key and end the program if it doesn't
:Stop
(...rest of code)
```

Simple enough

Thoughts to Consider

While discussing program protection, it is important to mention that somebody who's a knowledgeable calculator user will be able to circumvent any program protection using either one of the downloadable assembly programs that can unlock TI-Basic programs or the Graph Link software. Because of this, program protection really is only possible when you are dealing with ignorant calculator users.

Besides knowing about knowledgeable calculator users, you should also think about how others would be able to learn from your code. The general consensus among the calculator programming community is that programs should be unrestricted so others are able to study your work, as long as they do not release it as their own.

Putting all the code on one line would be frowned upon in this case because other programmers don't want to have to deal with separating out the code one line at a time to be able to understand and read it; that's just a major headache. Just remember that experimentation is key to learning TI-Basic, so you don't want to deprive that opportunity from someone else.

References

- David Martin had the ideas for "free trial" and "authorization" program protection in his TI-Basic guide, which unfortunately is not available on the Internet anymore. The examples given here are based on these ideas, but modified to fit this guide better.
- Weregoose came up with the plain password code example, while DarkerLine and Weregoose came up with the hash function password code examples; the examples were originally posted on the United-TI TI-Basic forum topic.

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/protection>

Look-Up Tables

A lookup table consists of a list (or matrix, depending on the situation) that is used to store calculations, for which the time to look them up in the list is smaller than having to calculate them (hence the name). Lookup tables are commonly created at or near the beginning of a program for later use in the program.

The primary advantage of lookup tables is their speed. Simply getting a number from a list is much faster than calculating the number with an algorithm or using a trigonometric function. The primary disadvantage of lookup tables is their memory usage. Not only do you need to use an extra variable to keep track of all the numbers, but it is very possible that you can end up storing numbers that you won't even use.

Examples

For an example, let's look at using the trigonometric functions. Say we want to draw a circle using lines. We want to draw a line every 10 degrees, and because the circle has 360 degrees, that means we would do 36 calculations. Without using lookup tables, the approach would be to simply use the `cos(` and `sin(` functions:

```
:ClrDraw
:0→Xmin:1→ΔX
:0→Ymin:1→ΔY
:90→A:30→B
:For(X,0,360,10
:70+20cos(X→C
:30+20sin(X→D
:Line(A,B,C,D
:C→A:D→B
:End
```

Although this code draws pretty fast already, it could be made faster using lookup tables. Every time through the loop we are calculating the cos(and sin(functions, which is quite time-consuming. Speed is especially important in this particular example because we want to have the circle draw as fast as possible (it should be faster than the Circle(command). Here's what the example would look like using lookup tables:

```
:ClrDraw
:0→Xmin:1→ΔX
:0→Ymin:1→ΔY
:90→A:30→B
:seq(20cos(X),0,360,10→L1
:seq(20sin(X),0,360,10→L2
:For(X,1,36
:70+L1(X→C
:30+L2(X→D
:Line(A,B,C,D
:C→A:D→B
:End
```

Another example that should help you more fully understand lookup tables is getting user input. More specifically, combining the getKey command with a lookup table. Say you want to display a text character based on which key was pressed. A common way to do this is to check to see what the individual keycodes are and then display the respective character:

```
:getKey→K
:If K=41
:Output(4,X,"A
:If K=42
:Output(4,X,"B
```

What you could do instead is place all the acceptable characters (in this case the alphabet) in a string, and then put all the keycodes in a list, organized to follow the alphabet keycodes. When you want to display a character, you simply search the list for the keycode that the user pressed:

```
:getKey→K
:{41,42,43,51,52,53,54,55,61,62,63,64,65,71,72,73,74,75,81,82,83,8
:If K>42 and K<94 and K≠45
:Output(4,X,sub("ABCDEFGHIJKLMNPQRSTUVWXYZ",max(K=Ans and seq(B,B
```

(See [Custom Text Input](#) page for a smaller and faster way to get the keycodes.)

Conclusion

These are rather simple examples, but they should be enough for you to understand how lookup tables work and what you can use them for. Just remember that the battle of speed vs. size is left up to you to decide which route you take. The two main factors to consider are the playability of the program (if the game is slow, the calculations should go) and the number of times the lookup table will be used (if the use is one, consider it none).

References

- The example trigonometric code was borrowed from David Martin's tutorial, which is not available on the Internet anymore.

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/lookuptables>

Self-Modifying Code (SMC)

Self-modifying code (SMC) is code that changes itself while it is executing. While TI-Basic does not provide support for SMC like it is found in other programming languages, TI-Basic does allow you to implement a primitive form of SMC using the [equation variables](#) that are used when graphing an equation on the graph screen.

The function, parametric, and polar graphing variables are accessible in the VARS menu (options 1, 2, and 3 respectively), while the sequence graphing variables are accessible on the keypad by pressing 2nd 7, 8, and 9 respectively.

Each of these variables is the same size, so there is no real difference between which variable you use. However, since sequence graphing is the least used graphing mode, the sequence variables are probably the best variables to use when using SMC.

Equation Variables	
Function	Y ₀ -Y ₉
Parametric	X/Y _{1T} -X/Y _{6T}
Polar	r ₁ -r ₆
Sequence	u, v, w

How does it Work?

While the equation variables are primarily used for graphing, they are actually stored internally as [strings](#) by the calculator. The string operations and commands do not work on them, however, but you can store a string to them and evaluate them in an expression.

Just like how you can evaluate an expression stored in a string using the [expr\(](#) command, the equation variables can be used the same way implicitly. The expression can contain whatever combination of numbers, variables, and functions that you want, just as long as they evaluate to a number or list of numbers.

For a simple example, let's initialize X with a value of 2 and store the expression "50X" to u:

```
: 2→X  
: "50X→u
```

When you access u , it will have a value of 100. But what would happen to the value of u if you changed the value of X to 5? Well, because the value of u depends on the value of X , u would change accordingly, and it would now have a value of 250. This is the basic premise of SMC: You can modify a variable elsewhere, and it will automatically update the respective equation variable. Hence, a program can change how it runs.

As it turns out, finding an occasion to use this technique is usually rare, so here is a made-up example. This program will count up and down with the arrow keys until you press ENTER. If you press 2ND, however, it will switch the order of the keys:

```
: 5→A  
: "(Ans=25) - (Ans=34→u"           // initial expression for u  
: Repeat Ans=105  
: A+u→A  
: Disp Ans  
: Repeat Ans:getKey:End           // wait for a keypress  
: If Ans=21  
: "(Ans=34) - (Ans=25→u"           // switch the arrow keys  
: End
```

Advanced Uses

While just using SMC for simple expressions doesn't really add any additional value to your programs, you can use it for more complicated purposes. One common situation is optimization.

If you have a lengthy command or formula that you use multiple times throughout your program, you can simply store the statement to an equation variable, and then use the equation variable whenever you want to call the statement. This eliminates duplicate code, which makes your program smaller.

For example, if you want to generate a random integer from 1 to 9, here is what you would use:

```
: "randInt(1,9→u"
```

Then each time you wanted to create a random integer, just use u .

Limitations of SMC

There are a few limitations you need to be aware of when using SMC:

- It complicates your code, making it difficult to understand and maintain. This is why you should primarily stick to implementing SMC when you are done with your program.
- The equation variables will affect the graph settings, and likewise the graph screen will be affected if the respective graph mode is enabled.
- You can't store the equation variable to itself, or other variables, if they don't have a matching type (i.e., trying to store a string to a real will result in an ERR:DATA TYPE error).
- Don't abuse SMC; the extra step of reading and executing through variables may slow down your code slightly and even cost a number of bytes if used improperly, so wield it wisely (i.e., only for the benefits it provides over other methods).

Grouping A Program

Your friend just asked you to transfer a program on your calculator over to his calculator so that he can play it in class whenever he wants to. You say sure, and he gets his link cable out and you start the transfer process. What you thought was going to be a simple transfer turns out to involve some serious headaches.

The program he wants not only has several subprograms that go with it, but multiple list and matrix variables as well as a few pictures. Unfortunately, you aren't aware of this until later after he tries to run the main program without the necessary subprograms, variables, and pictures.

When he asks you why the program won't work like it did on your calculator, you don't have an answer. You decide to start transferring over other programs and variables and whatever else to try to make the program work, but the program still doesn't cooperate. Finally you just give up and tell him that there must be something wrong with his calculator.

While this seems like a rather difficult problem to fix, there is in fact a simple solution: group files. Groups store files together in a package, where the file can be almost anything, whether it is a program, variable, picture, etc. Because groups reside in the archive, you never have to worry about a RAM clear deleting your group.

Advantages

Several files that go together can be put in one group, making it easy to transfer the files together — whether between two calculators, or a calculator and a computer.

On a TI-83+ or higher, group variables are stored in the archive, which means that a rogue RAM clear won't delete your files.

Groups are also great for backing up a version of a program being worked on before making major changes to it - even if the program is very large, or split between several files.

Finally, putting several parts of a released program in a group avoids the issue of users that forget to transfer a necessary file (although you should explain how groups work in a readme file).

Limitations

A group must contain more than one variable. It's possible to get around this by providing a dummy variable in the group (use a variable such as X that probably doesn't hold anything important).

TI-Basic programs cannot modify groups. You will have to recreate the group if you want to change its contents. Usually, this isn't too much trouble.

It's also been reported that in large enough groups, the calculator may change a bit in the data when ungrouping — in practice, this might result in an error when running the newly-ungrouped program. To be on the safe side, you should check that a group "works" before deleting the original files. It's also possible that splitting the large group in two (if this is feasible) will fix the issue.

References

- The idea for this grouping article came from Jon Pezzino and Kerm Martian's "The Elite Guide to TI-BASIC". It is a good read with lots of useful knowledge and tips/tricks. You can find the link to it on the [Resources](#) page.

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/grouping>

Subprograms



This is a featured article, which means it is one of the best articles available on TI-Basic Developer. A featured article not only is well-written, but includes everything about the topic, as well as references (if applicable). You can see more featured articles by going to the [featured articles](#) page.

Imagine you're creating an [RPG](#), and the current problem you're facing is how to display all of the data for each character on the screen. Without putting too much thought into it, the easiest approach to you seems to be to simply write out the data each time. Unfortunately, when you start implementing this approach, it soon becomes apparent that it will not work in your program.

While you've only programmed in a few of the characters and their respective data, you can already see that there is simply too much data to enter in; and then when you think about the possibility of having to go through and change all of the data again if you update the program, you're now left feeling overwhelmed and wondering if there's another way. Thankfully, there is, and its name is subprograms.

A subprogram is a program called by another program to perform a particular task or function for the program. When a task needs to be performed multiple times (such as displaying character data), you can make it into a separate subprogram. The complete program is thus made up of multiple smaller, independent subprograms that work together with the main program.

There are two different general types of subprograms available:

- **External** — They exist as stand-alone programs that are listed in the program menu, and can be executed just like a regular program.
- **Internal** — They are contained inside the program itself, so that they can be called by the program whenever needed.

There are certain advantages and disadvantages to using each type of subprogram, and consequently, situations where using one type of subprogram makes more sense than using the other. This doesn't mean you can't use whichever you like, though.

External Subprograms

External subprograms are the simplest type of subprogram, and involve executing one program from inside another program using the [prgm](#) command. You just insert the prgm command into the program where you want the subprogram to run, and then type the subprogram's name. You can also go to the program menu, and press ENTER on whichever program you want to use to paste the program's name into the program.

```
:prgmPROGNAME
```

When creating a subprogram, you take whatever code you want from the parent program and put it in its own program (see [your first program](#) for more information), and then put a [Return](#) command whenever you want to return to the parent program. (A Return command is optional at the end of a program, and you typically leave it off as part of [program optimization](#).)

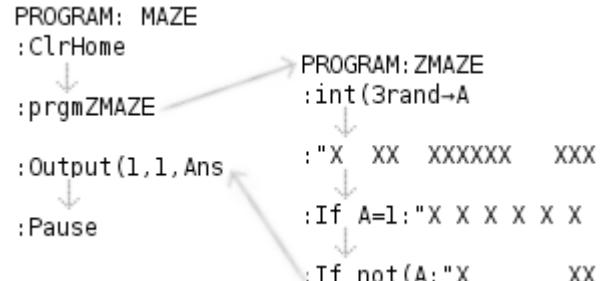
You should try to name your subprograms Zparentn or θparentn, where parent is the name of the parent program and n is the number (if you have more than one). Because subprograms are relatively unimportant by themselves, you want them to appear at the bottom of the program menu so they don't clutter it up and interfere with the user's ability to find whatever program they're looking for.

Here's a simple example where we are storing [maps](#) for our maze game in a subprogram, and then retrieving the desired map from the subprogram as a string, so we can print it out on the [home screen](#). (This example is somewhat contrived, but it should be enough to illustrate external subprograms).

```
PROGRAM:MAZE
:ClrHome
:prgmZMAZE
:Output(1,1,Ans
:Pause
```

```
PROGRAM:ZMAZE
:int(3rand→A
:"X      XX      XXXXXX      XX      XXXXXX      XX      XXXXX
:If A=1:"X X X X X X X X X X X X X X X X X X X X X X X X X X X
:If not(A:"X
                           XX
                           XX
                           XX
                           X
```

When the subprogram's name is encountered (in this case, prgmZMAZE), execution of the program will be put on hold, and program execution will transfer to the subprogram. Once the subprogram is finished running, or the Return command is executed, program execution will go back to the program, continuing right after the subprogram name. (If the [Stop](#) command is used instead of Return, however, the complete program hierarchy will stop.) See the image to the right for a graphical view of program flow using subprograms.



Although subprograms can call themselves (i.e., [recursion](#)) or other subprograms, each subprogram should return to the parent program and let it handle all the program calling. A structured program hierarchy is easier to understand and update (in fact, it's actually a [code convention](#)), and helps cut down the potential for [memory leaks](#).

Each program call is sixteen bytes, and gets put on a stack in RAM by the calculator. This is fine as long as its size isn't larger than the free RAM available, but each additional program call takes more memory until the program Returns. Having many nested program calls can run out of memory and crash the calculator (giving you a [ERR:MEMORY](#) error).

Passing Arguments

The main problem associated with trying to use external subprograms is that there is no built-in support for passing arguments to subprograms. This feature is important because it allows the subprogram to act differently based on what values are given to it for processing.

Fortunately, you can mimic this functionality by using variables. Because all variables are global (variables used by one program can be used by another), changing a variable in one program affects that variable everywhere else. While you can use almost any variable that you want, there are three main types of variables that you should choose from:

- **Pre-Defined Variables** — Includes reals, strings, matrices, built-in lists, etc. These variables are frequently used by most programs, so you don't have to worry very much about whether the user cares if you mess with them.
- **User-Defined Lists** — Uses the individual list elements to store different values. These variables have a certain sense of security that comes with using them because they are the only variable that you can actually create, so a program can have its own custom list to use.
- **Ans** — It can take on whatever value and variable you want, so the program doesn't have a specific variable hard-coded in. Its value changes whenever you store something or simply place an expression or string on a line by itself.

When using a pre-defined variable or user-defined list, you simply have to set the variable to the value that you want and then call the subprogram. The subprogram should be able to use that variable without problems, but if you don't properly setup the variable, the subprogram won't work correctly.

```
: {2,3,5,7,9→PRIME  
:prgmZPRIME
```

Using the Ans variable is essentially the same, except you need to add some additional code to the subprogram. Because the value of Ans changes whenever there is a new variable storage, you should have the first line inside the subprogram save Ans to another, more stable variable.

```
: {2,3,5,7,9  
:prgmZPRIME  
  
PROGRAM:ZPRIME  
:Ans→PRIME
```

This change saves some memory in the main program (in our example, we were able to get rid of the →PRIME statement), but the subprogram size is larger, since we really just shifted the variable storage code to the subprogram. However, if the subprogram is called multiple times this extra memory is only used once instead of once per call.

This does create a problem, though, because now when you store Ans to another variable, it will crash the program if Ans isn't the same type of variable. There is only one case where you can avoid crashing when the subprogram receives the wrong variable type. If the subprogram is expecting a real variable (such as A or X) and it is passed a list, it can prevent a crash by using the fact that a parenthesis ("") has multiple functions.

```
PROGRAM:ZSUB  
:Ans(1→A
```

The reason that this works is because a user-defined list doesn't need the L prefixed character

at the beginning when referring to the list. While you may be only asking the user to input a simple real variable, a list would also be allowed. There is nothing you can really do to fix this problem with other types, so just be careful.

Advanced Uses

The main consideration when using external subprograms is how many subprograms your program should have. While you're still putting your program together, it's good to keep it in many small, separate subprograms; but when you're done, all those subprograms become a liability and make your program unwieldy to use. This is because you have to remember all those subprograms in order to use your program.

There are two different ways to resolve this problem. The first way is to put the subprograms back in your program. This should only be done if a subprogram was only called once or twice, and putting it back in won't slow down the program. All you have to do is paste the code from the subprogram in place of the subprogram call. When you're done, you can delete the now unnecessary subprograms.

(The more detailed explanation is to go through your main program, and whenever you see a prgm call for a subprogram, clear that line and press 2nd STO. The Recall option will come up. Press the PRGM key and select the appropriate subprogram from the EXEC menu. The calculator will paste that subprogram's code into the main program.)

This is the same subprogram example from before, but now we've gotten rid of the ZMAZE subprogram and simply placed the subprogram code in the MAZE program itself:

```
PROGRAM:MAZE
:ClrHome
:int(3rand→A
:"X      XX      XXXXXX      XX      XXXXXX      XX      XXXXX
:If A=1:"X X X X X X X X X X X X X X X X X X X X X X X X X X X X
:If not(A:"X           XX           XX
:Output(1,1,Ans
:Pause
```

The second way to resolve the problem is by simply combining your subprograms together, so that there are fewer subprograms needed. Of course, how many subprograms you decide to use is up to you, but you should try to limit it to three or four subprograms at most, and just one subprogram ideally. This might take some effort on your part, but it will most certainly make your users happy.

When you start combining your subprograms together, you should place the subprograms one after the other, giving each subprogram a unique branching label (note that labels are local, so you can't use Goto in one program to jump to a label in another program). Instead of having to search through each individual subprogram, branching allows you to simply jump to the desired subprogram. You then just use the subprogram's variable argument to determine which subprogram to access.

```
:If A=e:Goto A // jump to first subprogram
:If A=π:Goto B // jump to second subprogram
:Stop // the subprogram was accidentally called
:Lbl A
: // subprogram code
:Return
:Lbl B
```

```
: // subprogram code  
: // No Return needed here
```

Looking at the example, the first thing you should notice is the variable values that are used to determine which subprogram to jump to. While you could use something simple like one or two, those values have a high probability of being accidentally entered in by the user or being set by another unrelated program. What works better is to use random decimals (like .193 or 1.857) or math symbols (like e or π).

If none of the variable values for the subprograms match, then none of the subprograms will be executed; instead, program execution will go to the Stop command on the next line, which will immediately stop the entire program. The reason for adding this program protection is to prevent the user from running the subprogram separate from our main program.

This is a real concern since external subprograms are listed in the program menu, and the user most likely at some point will try to run the subprogram just out of pure curiosity. Unless the user is a competent TI-Basic programmer who knows what they are doing, however, you normally don't want to let the user mess with your subprograms. If they change something, or delete some lines of code, then your program might stop working correctly.

The second thing you should notice about the example is the Return command at the end of each subprogram. If you have lots of subprograms, and you're accessing a subprogram near the bottom, it takes a considerable amount of time for program execution to go back to the main program. This happens because program execution doesn't return to the main program until after it reaches the end of the program, or it executes a Return command. So, just remember to include the Return commands as needed.

Advantages

There are several advantages of using external subprograms. First, and foremost, they reduce program size by eliminating redundant code. Instead of having to type the code multiple times for a task that occurs more than once in a program, you just type it once and put it in a subprogram. You then call the subprogram whenever you want to perform the task in your program.

Second, external subprograms increase program speed by making programs as compact as possible. You separate conditional tasks from the program (they either happen every time or they are skipped over), and put them in a subprogram; you then call the subprogram instead. This improves program speed because the calculator doesn't have to go through all of the conditional code anymore.

Third, external subprograms make editing, debugging, and optimizing easier. Instead of going through the entire program, looking for the code you want to change, you can focus on one subprogram at a time. This makes the code more manageable, allowing you to more thoroughly look at each subprogram and to better keep track of which subprograms you have looked at. It also prevents you from accidentally changing other parts of the program.

Lastly, subprograms are reusable, allowing multiple programs to share and use the same code. Breaking a program into smaller, individual subprograms, which each do a basic function or task, allows other programs to use those subprograms. Consequently, this reduces program size.

Internal Subprograms

Internal subprograms are the most complicated type of subprogram, and involve putting the subprograms in the main program itself. This is not the same thing as pasting the code from the

subprogram in place of the subprogram call, like you do with external subprograms; rather, it is designing your main program so that it can take advantage of subprograms, but all the code is self-contained.

There are several different ways that you can make internal subprograms, but the three most common ways are:

1. Append to the beginning of the program
2. Structured loops or branching
3. Branching out of broken loops

Append to Program Beginning

If you remember how we used external programs, then this should be very familiar. Instead of placing the subprograms in their own separate program, we are now just placing the subprograms at the beginning of our main program.

The standard way to make a subprogram is to use an If-Then conditional:

```
:If A=1.234:Then  
: // subprogram code  
:DelVar A  
:Return  
:End
```

Then to call the subprogram, you just set the variable to the desired value:

```
:1.234→A  
:prgmPRGNM
```

Of course, there are some important considerations that you need to be aware of. You can use whatever random value for the variable that you want, just as long as it isn't something that the user would typically use. This is to ensure that the subprogram isn't accidentally run when it shouldn't be, which is why you need to reset the variable's value inside the subprogram before exiting.

While you could use any variable that you want (including Ans), the best variables to use are the simple real variables (A-Z and θ). This is because they are small in size and they are constantly being used by other programs, so you don't have to really worry very much about your subprograms being accidentally run. (Ans is not a very good variable to use for the reasons listed above.)

You should always remember to include the Return command at the end of the subprogram. Once the subprogram is finished, the Return command will cause the subprogram to stop and return to the previous place in the program from where it was called. The other reason for the Return command is to prevent any memory errors that can occur if a program recursively calls itself too much.

Advanced Uses

You can have multiple subprograms at the beginning listed one after the other by simply using different values for the the variable:

```
:If A=1.234:Then
: // subprogram 1
:End
:If A=2.246:Then
: // subprogram 2
:End
```

While this works quite well when you only have three or four subprograms, with more subprograms it can actually slow down the main program. This happens because the calculator has to go through all the subprograms to get to the main program code.

You could fix this problem in a couple different ways, but the easiest way is to simply place all the subprograms in an If-Then conditional and then make that part of the subprograms. If this conditional is false, all of the subprograms will be skipped over.

A real number has an integer (accessed with the iPart(command) and fraction (accessed with the fPart(command) part, and you can use both of those for the subprograms: the integer will be the subprogram access check on the outside If-Then conditional and the fraction will be the respective subprogram we want to run.

```
:If 123456=iPart(A:Then // get integer part of number
:10fPart(A // get fraction part of number
:If Ans=1:Then
: // subprogram 1
:End
:If Ans=2:Then
: // subprogram 2
:End
: // rest of subprograms
:End
```

For calling the subprograms, you then just set the variable to the desired value like before:

```
:123456.1→A // run subprogram 1
:prgmPRGNAME
```

Structured Loops or Branching

If you don't like placing subprograms at the beginning of a program, the next approach that you can try is placing subprograms in the actual program code. While it would appear easy to simply place the subprograms wherever you feel like in your program, you can't readily do this since it would almost certainly cause your program to stop working correctly. Instead, you need to modify your program and subprograms so they can be put together.

What this modification entails is reorganizing your program so that the code works in a modular fashion as individual subprograms. This may not seem like it would be worth the effort, depending on the amount of code in your program, but modularization makes the program easier to understand and update (see planning programs for more information).

While there are several different ways you can structure the code in a modular fashion, the simplest way is to give each subprogram its own individual loop with a respective variable value as the loop condition. You can then access and exit the desired loop by simply changing the value of the variable. Of course, you have to determine which loop and variable you are going

to use.

There are three different loops you can choose from ([While](#), [Repeat](#), and [For\(\)](#)), but the best loop to use in this circumstance is While. This is because the condition is tested at the top of the loop, so if it's false already before the loop, then the loop will actually be skipped over (which is what allows us to use the loops as subprograms).

Once you have decided upon a particular loop, now you need to choose which variable you want to use. Like with the first way to make internal subprograms, the best variable to use is one of the real variables (A-Z and θ). This is because we just need a single value, and real variables only take up 15 bytes of memory (other variables are just as small, but they take up more memory when you're accessing them).

Now that the loop and variable have been chosen, we need to setup the system of loops to act as the subprograms. What works best is to have a main program loop and then place the subprogram loops inside of it. Putting the variable and loops together, here is what the program skeleton looks like:

```
:Repeat not(A // main program loop
:1→A
:While A=1
: // subprogram 1
:2→A // enter loop for second subprogram
:End
:While A=2
: // subprogram 2
:DelVar A // exit main program loop
:End
: // rest of subprograms
:End
```

You just set the value of the variable in the loop to use the desired subprogram. Then when you are done with the subprogram, you just change the value of the variable to something different to exit the loop. You do the same thing to exit the main program loop. You can use whatever system of values for the variable that you want, but just remember to keep it simple enough so that you can come back to it later and it still makes sense to you.

The one drawback of using this approach is that the calculator has to go through all the subprograms to exit the main program loop, which can really be slow depending on the size of the subprograms. At the same time, this approach is very easy to understand and follow because the loops are organized in a straight forward manner, so it's kind of an even trade off.

Related to using structured loops, the alternative you can use is branching. While using branching by itself to structure a program is generally frowned upon (see [planning programs](#) for more information), you can actually use it quite effectively for making internal subprograms that only need to be called a few times. Here is a simple example:

```
:θ→A:Goto A
:Lbl B
: // main program code
:1→A:Goto A
:Lbl C
: // main program code
:Stop
:Lbl A
: // subprogram code
```

```
:If A:Goto C  
:Goto B
```

The A variable is used for determining when to stop the program: a zero value will simply cause the subprogram to jump back to the main program, but a value of one will cause the subprogram to jump to the exit of the program (the C label). Because the calculator doesn't store the label positions, there is no way to get memory leaks using this approach, which is especially important when exiting the program. However, it does get hard to follow and maintain the code the more branching there is.

Branching out of Loops

The last way to make internal subprograms is arguably the most difficult to understand, but once you have it setup in your program, it provides an easy framework for adding additional subprograms. The best time to use these kind of subprograms is when you have a main program loop that you're running and you want to be able to jump out of it and then back into it whenever you want.

The basis of these subprograms is using branching ([Goto](#) and [Lbl](#)) with loops and [conditionals](#) (anything that uses an [End](#) command). Branching by itself allows the calculator to jump from one point in a program to another, skipping over whatever code you don't want executed. When you use branching to exit loops and conditionals, however, it has the unwanted effect of causing memory leaks.

Memory leaks happen because the calculator doesn't get to reach the End command for the associated loop or conditional, and the calculator just keeps on storing the End commands in its stack until there is eventually no free memory left. (Memory leaks can also occur with excessive program [recursion](#).) Here is a simple example that has a memory leak:

```
:Lbl A  
:While 1  
:Goto A  
:End
```

If you notice, when the Goto A command is executed, it jumps to the matching label A that is on the line before the loop. The While 1 loop is never allowed to finish because the End command never gets reached, and the branching occurs over and over again until the calculator finally slows down to a stop (because there is less and less free memory available) and returns a memory error.

This type of programming is common with beginners, and its use is generally frowned upon; instead you should try to use proper program structure (see [planning programs](#) for more information). However, if you know what you are doing, you can actually use these broken loops and conditionals for internal subprograms, and you won't have to worry about memory leaks or the dreaded memory error.

There are two different approaches that you can use. The first approach is to use another Goto and matching label to jump back into the loop. Because the calculator doesn't store the labels, you can freely use whatever branching in the loop that you want, and the calculator will act like it had never even left the loop:

```
:Repeat getKey  
:Goto A  
:Lbl B
```

```
:End  
:Stop  
:Lbl A  
: // subprogram code  
:Goto B
```

The key here is that the Goto A command jumps to the matching label A outside the loop, and then the Goto B jumps to the matching label B back inside the loop. The calculator still has the loop's associated End command on its stack, so the loop will just keep looping without problems until you eventually press a key to stop it and it executes the Stop command.

While this first approach works rather nicely in small programs, it is not very practical for use in large programs because all the branching starts to slow the program down. Unlike loops and conditionals, the calculator doesn't keep track of the label positions, so it must start from the beginning of the program to find the matching label to jump to. The further down the label is in the program code, the more time the calculator must spend looking for it.

The second approach solves this problem by using a duplicate End command for the loop or conditional. Since the calculator keeps track of the number of unfinished loops and conditionals by storing the associated End commands in its stack, we can make the calculator believe that our different End command is actually the End command that belongs to the loop. Here is a simple example to illustrate:

```
:Repeat getKey  
:Goto A  
:End  
:Stop  
:Lbl A  
: // subprogram code  
:End
```

Like with the first approach, when Goto A is executed the program will jump to the matching label A, and then the subprogram code will be executed. This time, however, the calculator will read the End command after the subprogram code, which it believes is the end of the loop, and then immediately jump back to the beginning of the loop. This process will be repeated over and over again until the user presses a key, at which time the Stop command will be executed and the program will stop.

The subprogram code for both approaches can be whatever you want, including other loops and conditionals. You just need to remember to close the loops and conditionals before returning to the original loop, otherwise the calculator will have the wrong End command on its stack. You also want to have a matching number of End commands for your loops and conditionals, or you will get a memory leak.

Advanced Uses

There are a couple different ways you can enhance the duplicate End subprogram approach so that you get the most use out of it. The first way is relatively simple, and just involves using a For(loop as the looping structure, instead of a While loop or Repeat loop (which is what we had in our previous examples).

A For(loop is basically a specialized form of a While loop, with the main differences being that it is executed a specific number of times and it has the variable setup and ending condition built-in. You just choose a variable, the range of values, and the increment (it is optional, with one as the default); and then the loop will start the variable at the initial value and increment it each

time until it reaches the ending value.

Now when you start using the For(loop for internal subprograms, you need to make sure the For(loop executes at least twice. This is so that the End command of the For(loop gets used along with the End command of the subprogram, otherwise it will simply fall through after the first time through the loop. You can select different subprograms based on the variable's value. Here is our example from above, now using a For(loop instead:

```
:For(A,0,1
:If not(A:Goto A
:End
:// main program code
:Stop
:Lbl A
:// subprogram code
:End
```

When the For(loop is executed, the A variable is set to zero. The not(A condition is true, so the calculator executes the Goto A command and then jumps to the matching label A. The calculator then jumps back to the beginning of the For(loop and increments the A variable to one. This time, however, there is no subprogram jump taking place, and the calculator simply finishes the For(loop like normal.

The second way to enhance the duplicate End subprogram approach is by using a simple trick. Because a While loop loops only when the condition is true, when the calculator comes across a While loop with a false condition, it will simply skip over the entire loop (and everything inside the loop). The easiest way to make a false condition is to use zero, since zero will never be true (based on Boolean logic). Here is a simple example to demonstrate:

```
:While 0
:Lbl A
:// subprogram code
:End
:// main program code
:If getKey:Then
:// program code
:Goto A
:End
```

When the calculator encounters the While 0 loop, it won't execute it because 0 is false. After the calculator enters the subprogram conditional, it executes some program code and then hits the Goto A command and jumps to the matching label A inside the While 0 loop. The calculator takes the End command of the loop as the End command of the conditional, and thus no memory leak occurs.

The reason that this trick is so valuable is because it allows you to place your subprograms at the beginning of the program, and you don't ever have to worry about them being accidentally executed (they will always be skipped over). In addition, now that the labels are at the beginning of the program, there is no more speed problem to deal with, since the calculator doesn't have to search through the entire program to find the labels.

Advantages

The main advantage of using internal subprograms is that there is only one program needed to

make your program run. When you give someone your program, you don't have to worry about forgetting to include any subprograms; or somebody deleting your subprograms afterwards, causing your program to stop working correctly. These things are mostly out of your hands, but users will think your program is at fault.

Related to the first advantage, the other advantage is that the user's program menu doesn't get cluttered up with insignificant subprograms. This problem is relative to how many subprograms a program has, but it can become tiresome to have to sort through the program menu in order to find the program that you want. If anything, this is just a nice courtesy to the users of your programs.

References

- Arthur O'Dwyer and his guide [The Complete TI-83 Basic Optimization Guide](#)
- Brad Wentz and his guide [Loop Trick for z80 Calcs](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/subprograms>

Multiplayer



This article is currently in development. You can help TI-Basic Developer by [expanding it](#). I will get around to finishing this, probably during the Summer Holidays. ~

James Kanjo

Multiplayer is where two or more people play a game together at the same time. Although you can create AI opponents for a player to play against, AI does not offer as much fun compared to playing against other human opponents — you can have the players work together, compete against each other, and even one player manage the other players.

Multiplayer games are generally divided into two categories: the players share a single calculator, and the players each play on their own calculator connected together with a link cable (either I/O or USB).

fold

Table of Contents

[Single-Calculator Multiplayer](#)

[Multi-Calculator Multiplayer](#)

[Core](#)

[One Screen](#)

[Two Screens](#)

[Final Notes](#)

[References](#)

Single-Calculator Multiplayer

Multiplayer games on one calculator generally fall into two categories: real-time and turn-based. Real-time is where the game action is constantly changing, not stopping for the player input. Some classic examples of real-time games are Galaxian and Pong. Turn-based is where each

player is allowed to make a move, and the game action only changes after each player has moved. Some classic examples of turn-based games are Battleship and Scorched Earth.

Making real-time games involves using the `getKey` command, except you can't wait for a key to be pressed. The general form is something along the lines of:

```
While <game not done>
getKey->K
If K=<player 1 key>:Then
// player 1 action
End
If K=<player 2 key>:Then
// player 2 action
End
<Update rest of game>
End
```

As you can see, the player action only occurs when a player has pressed a particular key; otherwise the game just continues on like regular, with the main game loop being repeated over and over again.

There are some problems with making real-time games, however.

The first, and foremost, problem is that TI-Basic is rather slow with user input. If you do anything remotely time-intensive, such as displaying lots of graphics or complex variable manipulation, then there will be some lag-time between when a player presses a key and when the calculator gets around to processing it. Although there's not really much you can do about this, you can make sure your game is as optimized as possible (especially breaking the more time-intensive parts into their own subprograms).

The second problem is that the calculator only keeps track of the last key pressed since the last time `getKey` was executed, so that means only one player can have their input read and processed each time through the game loop. In addition, if you enter a large block of code for the player, it will take a while before the other players have a chance to do anything.

Related to the second problem, the third problem with making real-time games is that unlike the other keys, the arrow and DEL keys can actually be held down, which will cause them to keep being repeated until they are unpressed. This effectively disables the other keys from being able to be pressed. There is no viable way to get around this problem, except asking the players to not press those keys.

The fourth problem is that the keypad is quite small, and having two or more people try to share the calculator can be rather difficult. The best way to work around this problem is choosing keys for each player that are a good distance away from each other. When choosing keys you should also keep in mind the calculator screen — if somebody has to press keys that make it difficult to see the screen, then you should choose different keys.

Because making real-time games is not very practical, a better alternative is turn-based games: you hand the calculator from player to player, allowing each player to move one at a time. These games are much easier to program: you simply use a variable to keep track of whose turn it is, increment the variable after each player's turn, and when everybody has completed their turn, you reset the variable. The only real downfall of turn-based games is that they can be slow because you have to wait until the other players are done before you can move.

Multi-Calculator Multiplayer

So I guess you're wondering how to program a multi-calculator multiplayer experience into one of your games. One of the first things you will need to do is familiarize yourself with the GetCalc(command. Basically, it retrieves a specified variable from another calculator and stores it to that variable on yours.

Creating multiplayer programs over two calculators is a much less simple process as it is to make single-calculator multiplayer programs. However, doing so could be the main selling point of your program and would certainly be worth the effort. You will notice that in each of our examples we tend to transfer lists, which we recommend you do too. Whilst it is possible to transfer a variety of real variables, it is much faster to transfer a list of numbers than a number of real variables.

There are two general ways to programming multi-calculator programs; one screen processing and two screen processing. The one screen processing method is simply making a program use the statistics from another calculator, and the whole multiplayer experience is processed on that calculator. The two screen processing method is much more complex, where we can share the multiplayer processing across two calculators by using a "turn-by-turn" interface.

Core

When it comes to multi-calculator multiplayer games, it is absolutely necessary to give each calculator its own identity. As both calculators are running the exact same program, we need a way to be able to determine one calculator as "Calculator A" and the other calculator as "Calculator B". This makes it possible for both calculators to know what data to send and receive. For example, if both calculators were "Calculator A", then both calculators would be doing exactly the same thing, or keep trying to receive the same variable from each other in an endless loop

This is code determines which assigns each calculator with a unique identity:

```
:GetCalc(A  
:e(A=π)+π(A≠π→A)
```

How it works is that the calculator gets the variable A from the other calculator, and checks whether it equals π (pi). If A equals π , then e is stored to A; however, if A does not equal π , then π is stored to A. Here is a table to demonstrate the results:

Calculator A	Calculator B
A = π	A = e
A = 3.141592654	A = 2.718281828

The calculator can therefore identify itself like this:

```
:If A=π:Disp "I'M CALC A  
:If A=e:Disp "I'M CALC B
```

If we use a not(routine to make Calculator A = 1 and Calculator B = 0 instead, then we are unable to determine whether a link has been initiated. Simply explained, because variable A is more likely to equal zero than any other number, Calculator A may accidentally assume Calculator B has initiated the multi-calculator sequence. Variable A is not likely to ever equal π (or e), which is why it's useful as a "connection initiated" checker for the calculator.

The beauty of the core code is that it doesn't matter which player executes the core code first, both calculators will be able to give themselves a unique identity, be able to distinguish which calculator they are and be able to see whether the other calculator is initiated yet.

One Screen

If you are looking to save space and valuable time, this is the multiplayer for you. This method has the sending calculator in a power-saving state the whole time while the receiving calculator does all of the hard work such as processing and animation.

+ See whole code

First off, we put in the core multiplayer code to determine which calculator is which. Then, Calculator B will retrieve the opponent's statistics for battling. Because the program uses the same variables on every calculator, we need to find a way to store Calculator A's statistics onto Calculator B without overwriting Calculator B's statistics. Surprisingly, this is not as hard as it seems:

```
:GetCalc(A  
:e(A=π)+π(A≠π→A  
:If A=π:Then  
:LSTATS→L1  
:Disp "SENDING DATA...", "PRESS ENTER WHEN  
:Pause "FINISHED  
:End
```

Now that each calculator has created its unique identity, and Calculator A has stored its statistics to L₁, we can finally make Calculator B receive Calculator A's statistics and process all of the data:

```
:If A=e:Then  
:GetCalc(L1  
<interactive code>  
:End
```

After this, write the rest as though this was a single-calculator multiplayer game, where you're statistics are in LSTATS and your opponent's statistics are in L₁. Here is a side-by-side comparison on how the program runs:

Calculator A	Calculator B

Two Screens

Here we show you a turn by turn based method of a battle game.

+ See whole code

Like with all multi-calculator multiplayer programs, we first provide the program with the core. This time, however, we will first reset variables A and F. Then we will add code for which stores each calculator's statistical data to its individually named list. Calculator B is then instructed to go elsewhere in the program (note that Goto is within an If-Then loop):

```
:DelVarADelVar FGetCalc(A  
:e(A=π)+π(A≠π→A  
:If A=π:Then  
:LSTATS→L1  
:Else  
:LSTATS→L2  
:Goto W  
:End
```

Because this is turn by turn battle game, we need to repeat the battle code until the battle is finished. We will make variable F determine this. Also, we want Calculator A to be able to attack first, so we shall put the code for attacking as the first thing in the Repeat loop. This is where it starts to get a bit sticky. First we delete variable B, which is going to determine what command the user has chosen (whether it be a kind of attack, or to run away).

```
:Repeat F  
:DelVar BMenu("CHOOSE ATTACK", "ATTACK A", A, "ATTACK B", B, "RUN AWAY", R  
:Lbl A:1→B:Goto S  
:Lbl B:2→B:Goto S  
:Lbl R:-1→B  
:Lbl S:1→θ
```

So when the user selects an option from the menu, a number is stored to B and 1 is stored to θ. Theta tells the receiving calculator that it the sending calculator is not ready yet. Then we create a second menu. This is to give the receiving calculator a chance to receive certain variables. That process is almost instant, and so the user then presses ENTER. θ is erased and 1 is stored to S, just before the "animation" program starts. S simply tells the animation program that it has just sent the attack.

```
:Menu("SENDING ATTACK", "READY?", SA  
:Lbl SA:1→S  
:DelVar θprgmθANIMAT
```

The animation program is shared by both the attacker and the attacked. The program makes particular animation depending on whether variable S is equal to one. If S=1, then this program will only display the opponent getting hurt. If S=0, then this program will display YOU getting hurt, calculate how much HP you have left, and if you died, sets variable F to zero. By using the subprogram "prgmθANIMAT", it means we don't need to worry about memory leaks or program changes.

Now we have the receiving code. You will notice that Lbl W is the first line here. This is so that Calculator B can jump straight here on the first move. Because this label is within a repeat loop, and the Goto came from an If-Then section, there are no memory leaks. The first thing we do is

make the program wait until the sending calculator has issued a move. If the game is over ($F=1$), then this process stops, and the program exits the "Repeat F" loop. If the game is not over, and the opponent issued a move, then the calculator receives the opponent's updated statistics.

```
:Lbl W:Disp "WAITING...
:DelVar BRepeat B or F
:GetCalc(B
:GetCalc(F
:End
:If not(F:Then
:If A=e:GetCalc(L1
:If A=π:GetCalc(L2
```

At the moment upon entering the loop, we know that the opponent's θ equals 1. In this loop, we clear our θ variable and then retrieve the opponent's θ . Because of the nature of GetCalc(), we can not receive variables whilst the other calculator is processing. Remember that as soon as the other calculator exits the "SENDING ATTACK..." menu, the animation subprogram starts.

Using this knowledge, if we delete θ and then retrieve θ whilst the opponent is in the "SENDING ATTACK..." menu, θ will equal 1 (and hence the loop is repeated). But when the opponent starts the animation process, we will be unable to retrieve θ , and so θ will equal what it all ready was, zero (hence we exit the loop).

```
:Repeat not(θ
:DelVar θGetCalc(θ
:End
:DelVar SprgmθANIMAT
:End
:End
```

This code is rather remarkable because it makes it possible to start the animation on both calculators, at the same time, automatically. You don't need to go messing about with "both users press ENTER at the same time" routines, only one user needs to press ENTER and both calculators begin — a foolproof technique.

If, after the animation process, the battle is not yet over, then the program continues into attack mode again (at the start of the "Repeat F" loop).

Now we need a routine which restores the statistics to the LSTATS list and says who won after the battle is over! This is the easiest part of the routine. If the player's health points do not equal zero, then that player won — otherwise the other player won.

```
:L1
:If A=e:L2
:Ans→LSTATS
:If Ans(6:Then
:Pause "YOU WON!
:Else
:Pause "YOU LOST!
:End
```

There are a couple of things that you need to be aware of for this routine to work:

1. When the "SENDING ATTACK" menu pops up, you must wait for a second for the attack to actually send before pressing ENTER;
2. Variable A must NEVER equal π during the program, (obviously with the exception of the multi-calculator code itself). If it does, and a different calculator starts this routine before the other one, then the link process will fail.

To prevent the second problem from ever happening, you should reset variable A at the start of your program, and never use this variable throughout the program.

This whole routine is certainly complex, but it does work, and works pretty well too. Here is a side-by-side comparison of how the program runs. For reference, here is a table showing the values:

	Calculator A	Calculator B
Max HP	$\lfloor \text{STATS}(1) = 2 \rfloor$	$\lfloor \text{STATS}(1) = 5 \rfloor$
Level	$\lfloor \text{STATS}(2) = 2 \rfloor$	$\lfloor \text{STATS}(2) = 5 \rfloor$
Attack	$\lfloor \text{STATS}(3) = 2 \rfloor$	$\lfloor \text{STATS}(3) = 5 \rfloor$
Defence	$\lfloor \text{STATS}(4) = 2 \rfloor$	$\lfloor \text{STATS}(4) = 5 \rfloor$
HP	$\lfloor \text{STATS}(6) = 2 \rfloor$	$\lfloor \text{STATS}(6) = 5 \rfloor$

Calculator A	Calculator B
	

Final Notes

Whilst the abilities of the GetCalc(command make it harder to create multi-calculator programs, it certainly is not impossible. You just need to think extra hard and create very clever workarounds to its boundaries.

If you have any questions or comments about these routines please ask in the discussion area for this page.

References

- James Kanjo came up with the "Multi-Calculator Core" code in his [IM](#) program, and also came up with the "turn by turn battle" code, including the function to make one calculator respond to another calculator's ENTER keypress

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/multiplayer>

Validation of User Input

Validation is the process of evaluating user input to ensure it satisfies the specified requirements. Most programs just blindly accept any input that the user enters, assuming that it was entered in correctly and that it is valid. This is a dangerous assumption, because there is a great likelihood that somebody at some point will enter in bad input. When that finally happens, the program will crash when it tries to use the input. In order to ensure this doesn't happen, you

should validate user input.

When validating user input, there are three main approaches you can take:

- **Stopping the program** — This validation approach involves asking the user for input, and then checking to see if it is a bad input value; if it is, you then just stop the program with a Return or Stop. The idea is that there is no point continuing on to the rest of the program, since the input would only cause the program to crash or get messed up when the input finally gets used.
- **Keep asking for input** — This validation approach involves placing your validation inside of a Repeat or While loop, and using a variable as a flag to indicate whether the input is valid or not. The idea is that after asking the user for input, you check to make sure it is not a bad input value; if it is, the bad input flag will be set, and input will be asked for again.
- **Make bad input valid** — This validation approach involves asking the user for input, and then trying to filter out any bad parts of the input that might be present. You can make the filters as in depth as you want, but because there is almost an infinite number of things that a user can enter in, there is no way to cover everything. Ultimately, it comes down to knowing the user, and what they are likely to enter in.

How to Validate Variables

There are three main variable types that a user will be asked to input values for in a program: reals, lists, and strings. Each of these variables has their own set of standard validation considerations to check for, along with whatever additional considerations you have in your particular program.

Reals

When validating a real variable, the main things to check for are:

- Is it within the appropriate range?
- Is it not a complex number (i.e., there is no imaginary *i* part)?
- Is it an integer (i.e., no fraction part)?
- Is it positive (or zero, if appropriate)?

Testing for these four conditions is relatively easy, considering that there are built-in commands that will take care of it for you: relational operators, the imag(command, and the fPart(command. Generally, you should do the complex number check before the other three checks, because an imaginary part in a number can cause problems.

There's another problem that comes up with using Input specifically. If the input is for a real number A, the user might enter a list, which will be stored to LA instead. A simple way of fixing this problem is to store an invalid value to A to begin with, which will be dealt with if the user enters a list and doesn't change A. Many times, this is as simple as using DelVar.

For a simple example of validating a real number, imagine asking the user to specify how many rounds of a game they want to play. A For(loop is used as the main game loop, with the number of rounds being the upper boundary. Since the lower boundary is zero, we want the user to input a number greater than zero. In addition, a For(loop does not work with complex numbers, and we do not want to allow a fraction part because the number of rounds is used to calculate the scoring.

Here is what the validation code for our game might look like with each of the three different validation approaches:

:DelVar AClrHome	:0→A	:DelVar AClrHome
:Input "ROUNDS: ",A	:Repeat Ans	:Input "ROUNDS: ",A
:If imag(A	:ClrHome	:max(1,iPart(real(A→A
:Return	:Input "ROUNDS: ",A	
:If A<1 or fPart(A	:If not(imag(A	
:Return	:not(A<1 or fPart(A	
	:End	

There are a couple things you should note. In the stopping the program code, we used the Return command to stop the program, instead of the Stop command. This is so that the program will work correctly with subprograms and assembly shells. We used the opposite commands in the making bad input valid code: iPart(instead of fPart(and real(instead of imag(. We are also using the max(command to make one the minimum value for the input.

Lists

When validating a list, the main things to check for are:

- Is the list length within the appropriate range?
- Does each list element pass the real validation?

Testing for the list length condition and each of the list elements involves using the built-in dim(command. You first check to see that the list length is acceptable, and then use the dim(command as the upper boundary in a For(loop to go over the list elements one at a time. Each element is validated just like a real variable.

For a simple example of validating a list, imagine you have a lottery game and you want the user to specify three numbers. We want the numbers to be between 1-100, as well as not having an imaginary or fraction part. Here is what the validation code for our game might look like with each of the three different validation approaches:

:ClrHome	:Repeat A=3
:Input "NUMBERS: ",L1	:ClrHome
:If 3≠dim(L1:Return	:Input "NUMBERS: ",L1
:For(I,1,3	:DelVar A
:L1(I	:For(I,1,3(3=dim(L1
:If imag(Ans:Return	:L1(I
:If Ans<1 or Ans>E2 or fPart(Ans	:If not(imag(Ans
:Return	:A+not(Ans<1 or Ans>E2 or fPart(Ans→A
:End	:End:End

Like with the example from before, we had to check for the complex number before checking for the number boundaries and fraction part. This is because neither of those commands work with complex numbers; they will actually throw a ERR:DATA TYPE error. Also important is the optimization that we used to move the list dimension check into the For(loop's upper boundary. This allowed us to eliminate a conditional that we would have had to add.

Strings

When validating a string, the main things to check for are:

- Is the string length within the appropriate range?
- Does the string only contain the appropriate characters?

Testing for the string length involves using the built-in `length` command. This check by itself is not enough, however, because a string treats commands and functions as just one character (i.e., a string of "ABOutput(" is considered to be three characters long). The way you resolve this problem is by making sure the string only contains certain characters. This involves creating a string of acceptable characters, and then checking the user input against it.

For a simple example of validating a string, imagine you have a two-player hangman game and you want the user to enter in an eight letter word, so that the other player can guess it. The only characters that are allowed are the uppercase alphabet (A-Z), and there is no restriction that the word has to actually exist. (Programming in a check for that would involve keeping a dictionary of words, and that could potentially take up a lot of memory.)

Here is what the validation code for our game might look like with each of the three different validation approaches:

```
: "ABCDEFGHIJKLMNPQRSTUVWXYZ
:ClrHome
:Input "WORD: ",Str1
:If 8≠length(Str1:Return
:If not(min(seq(inString(Ans,sub(Str1,I,1))),I,1,8
:Return
: "ABCDEFGHIJKLMNPQRSTUVWXYZ
:0:Repeat Ans
:ClrHome
:Input "WORD: ",Str1
:If 8=length(Str1
:min(seq(inString(Str1,I,1,8
:End
```

When the user inputs a word, we loop through all of the characters in the word and get their positions in our acceptable characters string. If any of the characters weren't in the string, then their position will be zero, and when we take the minimum of all the positions, the smallest will be zero. With the making bad input valid code, we also concatenate how ever many characters we need to make the word eight characters long.

Making Validation More Friendly

There are a couple different ways you can make validation more user-friendly: displaying error messages when there is bad input, and storing input as a string and converting it to the appropriate variable type.

Displaying error messages to the user when they enter bad input helps the user correct their mistake, and provides some direction on what input you are expecting them to enter. An error message does not need to be complicated or long — just enough so that you can get the point across. For example, say a program is a number guessing game, and the user is expected to enter in a number between 1-1000. If they enter 5000, you can display the message "Between 1-1000".

Storing input as a string allows you to accept any input that the user may enter, even if it is messed up, entered in in the wrong format, or inappropriate for the variable that you are storing it to. This way instead of the program crashing when it gets bad input, it can actually handle it and do whatever it needs to to make it work. You then just check to see if the string has the appropriate value(s), and convert it to the desired variable using the `expr` command.

The validation for the real variable, for example, did not include a check for whether it is a list or string. This is because you can only really check for those things when you have a string that you can manipulate. If we wanted to add that check, we can search the string for an opening

curly brace and commas or any characters besides numbers. If we find those things, we know that the input is bad, and we can reject it and ask for input again.

In the case of the real variable, the other advantage of using a string is that you don't have to worry about whether the calculator is storing the variable to a list. More specifically, if a list is entered for input, the Input and Prompt commands will actually store the input to a list with the same name. This is possible because you don't need to include the L character when referring to a user-defined list. (Entering a string would also work, and the string would become associated with the list.)

Besides checking the list for whether it is a real variable or string, you also can check that it is in the appropriate format. When a list is entered, it needs to start with an opening curly brace, and then have each element separated by a comma. Because most users forget to include the opening curly brace, it is very convenient to place that at the beginning of the list yourself, so that the user never even knows about needing it.

You can take that idea even further, and allow a list to be entered in many different ways: with commas and curly brackets, with commas and no brackets, as a list name (L1 through L6), or as a name starting with L, or as a name without the L. Instead of requiring one of these, the program might very well be programmed to handle all of them. This all comes down to finding alternate ways of making user input valid.

Thoughts to Consider

The amount of validation you put in a program depends on how large and/or complicated the program is. If you have an extremely complex game with all sorts of user input, then it would be appropriate to include validation for some or most of those things. The general guideline is that the amount of validation needed correlates to the size of the game — i.e., a short math routine probably wouldn't need validation.

While discussing validation, it is also important to mention that since the Input and Prompt commands only work on the home screen, you need to write your own custom input routine using the getKey command if you want to get input on the graph screen. If your entire program is already on the graph screen, however, this should not be an issue, because it makes perfect sense to maintain the user's attention on the graph screen.

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/validation>

Recursion

Good programmers usually design their programs to utilize subprograms (calling another program from within the program) for optimization but another alternative that is available, but less often used, is the program simply calling itself — more commonly known as recursion.

The basic premise behind recursion is breaking up a problem into smaller problems, and then working your way through each problem until they are all completed. By tackling one small problem at a time, instead of the entire problem, the code needed is typically not only smaller and easier to understand (i.e., more manageable), but also tends to be faster.

However, recursion isn't always the most appropriate approach. You can usually rewrite a recursive program to use iteration instead (whether it's a While, Repeat, or For(loop). While the iteration code may be larger, it doesn't need the additional memory for each call that the program makes like recursion does. Iteration is also better when trying to implement an algorithm with recursion isn't very practical.

Problems with Recursion

There are some problems you will come across when trying to use recursion in your programs. Each of these problems is inherent to TI-Basic because of the way TI designed it, which means you can't change them. Fortunately, you can use some creative thinking to work around them.

The first problem you will come across is that you can only call a program a set number of times before you run out of memory and the program crashes — giving you the dreaded ERR:MEMORY error. The reason that this happens is because the calculator places each program call on a stack.

The program call stack is kept in RAM, so it is fine as long as its size doesn't exceed the amount of free RAM available. Each program call takes up approximately sixteen bytes, so just divide that by the free RAM to see how many program calls you can make.

Besides simply limiting the number of program calls you make in a program (i.e., trying to keep recursion to a minimum), a work around to this problem is storing a special value to a variable (something unique that wouldn't be entered by accident), displaying a message to the user telling them to "Press ENTER" and then stopping the program with the Return command after a set number of program calls have occurred.

```
:312958→A  
:Output(4,4,"Press ENTER  
:Return
```

Once the user presses ENTER, you will want to include a check at the beginning of the program for the variable you used to see if its value is equal to the unique value you assigned it. If it is, you then can jump to the place in the program where you left off before. You also want to give the variable a new value so that the program won't accidentally jump to the place in the program when the program is next executed.

```
:If A=312958 // Check if variable equal to unique value  
:Goto A  
...  
:Lbl A  
:1→A // Reset variable to a new value
```

Another problem you will cross across is that TI-Basic programs don't have return values. In 68k TI-Basic (which is much more powerful overall), a return value can be passed to the calling program, which can then use it however they want (for example, to determine which course of action to take next).

While you can't add a return value to a program, you can mimic that functionality using a variable. The best variable to use is Ans because it can take on whatever value and variable type you want, so the program doesn't have a specific variable hard-coded in. This is especially important because variables are shared by every program.

Related to creating a return value is the problem of creating (pseudo) local variables. As you deal with each program call in recursion, it is useful to be able to keep track of variables and how they change from one program call to the next. While there are no local variables, you can make a list perform in that capacity.

In addition to the list itself, an index variable that keeps track of where you are in the list is also required. Whenever you enter a program that needs a local variable, increase the index variable

and add a new element to the end of the list with `augment(` \backslash NAME,{var}). When you exit the program, decrease the index variable and remove the element from the list with `var \rightarrow dim(` \backslash NAME). You can access the local variable at any time with `\backslash NAME(var)`.

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/recursion>

Assembly

Besides TI-Basic, assembly is the other primary programming language available for the TI-83 series calculators. Unlike TI-Basic, which uses commands and functions that are easy to understand, assembly is programmed in the calculator's own machine language. Thus, it is much harder to program in and read.

This lack of usability, however, is more than made up for when you consider the fact that assembly is much faster and more feature rich than TI-Basic. Games that normally can't be done (or, if they can be done, they aren't done very well) in TI-Basic are just considered average in assembly.

At the same time, there aren't as many assembly programmers compared to TI-Basic programmers; and subsequently, effort has been made to enhance TI-Basic using assembly, to make it more capable of quality games and programs. This includes:

- **Shells** — A shell allows a person to run a program from inside one central place, and since most assembly programs are run through a shell, it makes sense to run your TI-Basic programs there as well.
- **Libraries** — A library enhances a TI-Basic in some way, providing support to an internal function of the calculator (such as lowercase text) or access to a peripheral of the calculator (such as the USB port on the TI-84+/SE).
- **Applications** — Basic Builder allows you to package your TI-Basic program(s) into a Flash application, which appears in the APPS menu and gets executed just like a regular assembly application.

The TI-Basic language itself provides three commands — `Asm()`, `AsmPrgm`, and `AsmComp()` — for running and compiling shell-independent assembly programs, which you simply run from the home screen or inside a program. Writing these kind of assembly programs is actually more difficult, however, because the assembly language instructions are represented as hexadecimal numbers.

Two additional commands for running assembly programs have been added on the TI-84 Plus and TI-84 Plus SE calculators: `OpenLib()` and `ExecLib`. They can be used for running routines from Flash application libraries that have been specifically written for use with these commands. So far, however, most major libraries use other methods, for compatibility with pre-TI-84 calculators. The only existing software that uses `OpenLib()` and `ExecLib` is `usb8x`, a library for advanced use of the TI-84 Plus/SE USB port. Here, compatibility is obviously out of the question.

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/assembly>

The $10^{\wedge}($ Command

The $10^{\wedge}($ command raises 10 to a power. Since it's possible to just type out 1, 0, $^{\wedge}$, and $($, the reason for

`10^(\theta)`

having a separate function isn't immediately obvious, but the command is occasionally useful.

`10^(` accepts numbers and lists as arguments. It also works for complex numbers.

```
10^(2)  
100  
10^{ {-1, 0, 1} }  
{ 0.1 1 10 }
```

Optimization

Don't type `10^(` out, use this command instead. It's three bytes smaller and usually faster as well. However, keep in mind that you might be able to use the `E` command instead of `10^(`, for constant values.

Command Timings

The command `10^(` is faster than typing out `10^(` in most cases, except for small integer arguments. Even faster is `E`, but that only works for raising 10 to a constant power.

Related Commands

- `E`
- `log(`
- `^`

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/ten-exponent>

The `^-1` Command

The `^-1` command returns the reciprocal of a number, equivalent to dividing 1 by the number (although reciprocals are sometimes more convenient to type). It also works for lists, by calculating the reciprocal of each element.

The `^-1` command can also be used on matrices, but it is the matrix inverse that is computed, not the reciprocal of each element. If $[A]$ is an N by N matrix, then $[A]^{-1}$ is the N by N matrix such that $[A][A]^{-1} = [A]^{-1}[A]$ is the identity matrix. `^-1` does not work on non-square matrices.

```
10^(1)  
10  
10^{ {-1, 0, 1, 2} }  
{ .1 1 10 100 }
```

Command Summary

Raises 10 to a power.

Command Syntax

`10^(value)`

Menu Location

Press [2nd] [10^x] to paste `10^(`.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

```
2^-1  
[[3, 2] [4, 3]]^-1  
[[3, -2]  
[-4, 3]]
```

Command Summary

Returns the reciprocal of a number (1 divided by the number). For matrices, finds the matrix inverse.

Command Syntax

```
{1,2,3}¹
{1 .5 .3333333333}
[[3,2][4,3]]¹
[[3 -2]
[-4 3]]
```

Much like the number 0 does not have a reciprocal, some square matrices do not have inverses (they are called singular matrices) and you'll get an error when you try to invert them.

Optimization

Writing $A^{-1}B$ instead of B/A is sometimes beneficial when B is a complicated expression, because it allows you to take off closing parentheses of B . For example:

```
: (P+√(P²-4Q))/2
can be
: 2¹ (P+√(P²-4Q)
```

This may be slower than dividing. There are also situations in which this optimization might lose precision, especially when the number being divided is large:

```
7fPart(4292/7
      1
7fPart(7¹4292
      .999999999
```

Error Conditions

- **ERR:DIVIDE BY 0** is thrown when trying to take the reciprocal of 0.
- **ERR:SINGULAR MAT** is thrown when trying to invert a singular matrix.

Related Commands

- $\frac{2}{3}$

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/inverse>

The 1-PropZInt(Command

The 1-PropZInt(command calculates a confidence interval for a proportion, at a specific confidence level: for example, if the confidence level is 95%, you are 95% certain that the proportion lies within the interval you get. The command assumes that the

value⁻¹

Menu Location

Press [x^{-1}]

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

```
1-PropZInt
(.46901,.53099)
̂=.5
n=1000
```

sample is large enough that the normal approximation to binomial distributions is valid: this is true if, in the sample you take, the positive and negative counts are both >5 .

The 1-PropZInt(command takes 3 arguments. The first, x , is the positive count in the sample. The second, n , is the total size of the sample. (So the sample proportion is equal to x out of n). The third argument is the confidence level, which defaults to 95.

The output gives you a confidence interval of the form (a,b) , meaning that the true proportion π is most likely in the range $a < \pi < b$, and the value of x/n .

Sample Problem

You want to know the proportion of students at your school that support a particular political candidate. You take a random sample of 50 students, and find that 22 of them support that candidate. 22, the positive count, and $50-22=28$, the negative count, are both >5 , so the assumption is satisfied.

Using 22 for x , and 50 for n , you decide to find a 95% confidence interval. The syntax for that is:

```
:1-PropZInt(22,50,95  
which can also be  
:1-PropZInt(22,50,.95)
```

The output if you run the above code will look approximately like this:

```
1-PropZInt  
(.30241,.57759)  
p=.44  
n=50
```

This tells you that between about 30.2% and about 57.8% of the students at your school are in support of the political candidate.

Optimization

If the confidence level is 95%, you can omit the final 95, since that is the default value:

```
:1-PropZInt(22,50,95  
can be  
:1-PropZInt(22,50)
```

Command Summary

Computes a Z confidence interval of a proportion.

Command Syntax

`1-PropZInt(x , n [, confidence level])`

Menu Location

When editing a program, press:

1. STAT to access the statistics menu
2. LEFT to access the TESTS submenu
3. ALPHA A to select 1-PropZInt(, or use arrows

(this key sequence will give you the 1-PropZInt... screen outside a program)

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

Error Conditions

- **ERR:DOMAIN** is thrown if the sample proportion is not between 0 and 1, any argument is negative, or the confidence level is 100 or more.

Related Commands

- [2-PropZInt](#)
- [ZInterval](#)
- [2-SampZInt](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/1-propzint>

The 1-PropZTest(Command

1-PropZTest performs an z-test to compare a population proportion to a hypothesis value. This test is valid for sufficiently large samples: only when the number of successes (x in the command syntax) and the number of failures ($n-x$) are both >5 .

The logic behind the test is as follows: we want to test the hypothesis that the true proportion is equal to some value p_0 (the null hypothesis). To do this, we assume that this "null hypothesis" is true, and calculate the probability that the (usually, somewhat different) actual proportion occurred, under this assumption. If this probability is sufficiently low (usually, 5% is the cutoff point), we conclude that since it's so unlikely that the data could have occurred under the null hypothesis, the null hypothesis must be false, and therefore the true proportion is not equal to p_0 . If, on the other hand, the probability is not too low, we conclude that the data may well have occurred under the null hypothesis, and therefore there's no reason to reject it.

Commonly used notation has the letter π being used for the true population proportion (making the null hypothesis be $\pi=p_0$). TI must have been afraid that this would be confused with the real number π , so on the calculator, "prop" is used everywhere instead.

In addition to the null hypothesis, we must have an alternative hypothesis as well - usually this is simply that the proportion is not equal to p_0 . However, in certain cases, our alternative hypothesis may be that the proportion is greater or less than p_0 .

The arguments to 1-PropZTest(are as follows:

```
1-PropZTest
PROP=.5
z=-3.4
P=6.7396165E-4
P=.33
n=100
```

Command Summary

Performs a z-test on a proportion.

Command Syntax

`1-PropZTest(p_0 , x , n [, alternative, draw?])`

Menu Location

While editing a program, press:

1. STAT to access the statistics menu
2. LEFT to access the TESTS submenu
3. 5 to select 1-PropZTest, or use arrows

(outside the program editor, this will select the 1-PropZTest... interactive solver)

Calculator Compatibility

TI-83/84/+/SE

Token Size

- p_0 - the value for the null hypothesis (the proportion you're testing for)
- x - the success count in the sample
- n - the total size of the sample (so the sample proportion would be x/n)
- *alternative* (optional if you don't include *draw?*) - determines the alternative hypothesis
 - 0 (default value) - $\text{prop} \neq p_0$
 - -1 (or any negative value) - $\text{prop} < p_0$
 - 1 (or any positive value) - $\text{prop} > p_0$
- *draw?* (optional) set this to 1 if you want a graphical rather than numeric result

2 bytes

Although you can access the 1-PropZTest command on the home screen, via the catalog, there's no need: the 1-PropZTest... interactive solver, found in the statistics menu, is much more intuitive to use - you don't have to memorize the syntax.

In either case, it's important to understand the output of 1-PropZTest. Here are the meanings of each line:

- The first line, involving "prop" and p_0 , is the alternative hypothesis.
- z is the test statistic. If the null hypothesis is true, it should be close to 0.
- p is the probability that the difference between the proportion and p_0 would occur if the null hypothesis is true. When the value is sufficiently small, we reject the null hypothesis and conclude that the alternative hypothesis is true. You should have a cutoff value ready, such as 5% or 1%. If p is lower, you "reject the null hypothesis on a 5% (or 1%) level" in technical terms.
- \hat{p} is the sample proportion, x/n .
- n is the sample size.

Advanced Uses

The final optional argument of 1-PropZTest, *draw?*, will display the results in a graphical manner if you put in "1" for it. The calculator will draw the standard normal distribution, and shade the area of the graph that corresponds to the probability p . In addition, the value of z and the value of p will be displayed. You would make your conclusions in the same way as for the regular output.

Optimization

Some of the arguments of the 1-PropZTest command have default values, and the argument can be omitted if this value is used.

- The *draw?* argument can be omitted if you don't want graphical output, although you could put "0" in as well.
- If the above argument is omitted, and you're doing a two sided test, you may omit the *alternative* argument.

Example:

```
:1-PropZTest(.5,22,50,0,0
can be
:1-PropZTest(.5,22,50
```

Error Conditions

- **ERR:DOMAIN** is thrown if p_0 or x/n are not between 0 and 1, or x is negative or greater than n (however, any real value for *alternative* and *draw?* will work)

Related Commands

- 2-PropZTest(
- Z-Test(
- 2-SampZTest(

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/1-propztest>

The 1-Var Stats Command

This command calculates a bunch of common (and a few uncommon) statistics for a list (it uses L1 by default, but you can use any list by supplying it as an argument). You have to store the list to a variable first, though, before calculating statistics for it. For example:

```
: {5,12,7,8,4,9→L1
:1-Var Stats
```

Like other statistical commands, you can use a frequency list as well, for cases where one element occurs more times than another (you can do this with a normal list, too, but that might be inconvenient when an element occurs very many times). For example:

```
: {1,2,3→L1
: {5,3,2→L2
:1-Var Stats L1,L2
```

is the frequency-list equivalent of:

```
: {1,1,1,1,1,2,2,2,3,3→L1
:1-Var Stats
```

When you're running it from the home screen, 1-Var Stats will display the statistics; this won't happen if you do it inside a program. Either way, it will also store what it calculated to the statistics variables found in VARS>Statistics... The variables 1-Var Stats affects are:

- \bar{x} is the mean (average) of the elements, as returned by mean(

```
1-Var Stats
 $\bar{x}=3$ 
 $\Sigma x=15$ 
 $\Sigma x^2=55$ 
 $Sx=1.58113883$ 
 $\sigma x=1.414213562$ 
 $\downarrow n=5$ 
```

Command Summary

Calculates some statistics for a single list of data, and stores them to statistical variables. They're also displayed in a scrollable list, if done outside a program.

Command Syntax

1-Var Stats [*list*, [*freqlist*]]

Menu Location

Press:

1. STAT to access the statistics menu
2. LEFT to access the CALC submenu
3. 1 or ENTER to select 1-Var Stats

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

- Σx is the sum of the elements, as returned by sum()
- Σx^2 is the sum of the squares of the elements
- Sx is the sample standard deviation, as returned by stdDev()
- σx is population standard deviation
- n is the number of elements in the list, as returned by dim()
- $\min X$ is the minimum value, as returned by min()
- $Q1$ is the first quartile
- Med is the median, as returned by median()
- $Q3$ is the third quartile
- $\max X$ is the maximum value, as returned by max()

1-Var Stats will not work with "reserved" list names that the calculator uses internally. The only known such reserved list is the list RESID, and there's no reason to suspect there are any others. Ans, TblInput, and any expression which resolves to a list, are also not appropriate for this command: store all of these to a list before doing 1-Var Stats on them.

Optimization

Aside from statistical analysis, 1-Var Stats can also be used when you want to use the values it calculates more than once. This will save on size, since, for example Σx takes up less space than $\text{sum}(L1)$, but considering how many calculations 1-Var Stats makes, it will usually be slower. Here's a short example which saves 1 byte:

```
:Disp "RANGE:",max(L1)-min(L1
can be
:1-Var Stats
:Disp "RANGE:",maxX-minX
```

Related Commands

- [2-Var Stats](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/1-var-stats>

The 2 Command

The 2 command raises an input to the second power. It has exactly the same function as " 2 ", but is one byte smaller. If used on a list, it will return a list with all of the elements squared. If used on a matrix, it will return the second matrix power of the input matrix.

```
22
        4
{1, -2, 3}2
        {1 4 9}
[[2, -1] [-3, 0]]2
        [[1 -2]]
```

```
22
        4
[[1,2][2,4]]2
        [[5 10]
[10 20]]
```

Command Summary

Raises the input to the second power.

Optimization

Use this command instead of $\wedge 2$ in all instances.

```
:X^2
can be
:X2
```

Related Commands

- $\frac{-1}{3}$
- $\frac{3}{3}$

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/2>

The 2-PropZInt(Command

The 2-PropZInt(command calculates a confidence interval for the difference between two proportions, at a specific confidence level: for example, if the confidence level is 95%, you are 95% certain that the difference lies within the interval you get. The command assumes that the sample is large enough that the normal approximation to binomial distributions is valid: this is true if, in both samples involved, the positive and negative counts are both >5 .

The 1-PropZInt(command takes 5 arguments. The first two, x_1 and n_1 are the positive count and total count in the first sample (so the estimated value of the first proportion is x_1 out of n_1). The next two arguments, x_2 and n_2 , are the positive count and total count in the second sample.

The output gives you a confidence interval of the form (a,b) , which is the range of values for the difference $\pi_1 - \pi_2$ (where π_1 and π_2 are the first and second proportions respectively). If you were looking for the difference $\pi_2 - \pi_1$ all you have to do is switch two sides and negate the numbers in the interval.

Sample Problem

You want to compare the proportion of students at your school and at a friend's school. that support a

Command Syntax

*value*²

Menu Location

Press [x^2]

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

```
2-PropZInt
(-.2023,.00227)
P1=.45
P2=.55
n1=100
n2=1000
```

Command Summary

Computes a Z confidence interval of the difference between two proportions.

Command Syntax

2-PropZInt($x_1, n_1, x_2, n_2, \underline{\text{level}}$)

Menu Location

When editing a program, press:

1. STAT to access the statistics menu
2. LEFT to access the TESTS submenu
3. ALPHA B to select 2-PropZInt(, or use arrows

(this key sequence will give you the

particular political candidate. You take a random sample of 50 students, and find that 22 of them support that candidate. Your friend took a random sample of 75 students at his school, and found that 28 supported the candidate.

The first proportion is the proportion of supporters at your school. 22 out of 50 students support the candidate, so $x_1=22$ and $n_1=50$.

The second proportion is the proportion of supporters at your friend's school. 28 out of 75 students support the candidate, so $x_2=28$ and $n_2=75$.

If you decided to do a 95% confidence interval, you would add the argument 95 after all these, so the syntax would be as follows:

```
:2-PropZInt(22,50,28,75,95  
which can also be  
:2-PropZInt(22,50,28,75,.95)
```

The output if you run the above code will look approximately like this:

```
1-PropZInt  
(-.1092,.24249)  
p1=.44  
p2=.3733333333  
n1=50  
n2=75
```

This tells you that between about the difference between the proportions is between about -0.11 (your school's proportion being about 0.11 less than your friend's school's proportion) to about 0.24 (your school's proportion being about 0.24 greater than your friend's school's proportion).

Optimization

If the confidence level is 95%, you can omit the final 95, since that is the default value:

```
:2-PropZInt(22,50,28,75,95  
can be  
:2-PropZInt(22,50,28,75)
```

Error Conditions

- **ERR:DOMAIN** is thrown if either proportion is not between 0 and 1, or x_i is negative or greater than n_i , or the confidence level is negative or at least 100.

Related Commands

- **1-PropZInt**

- ZInterval
- 2-SampZInt

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/2-propzint>

The 2-PropZTest(Command

2-PropZTest performs an z-test to compare two population proportions. This test is valid for sufficiently large samples: only when the number of successes (x in the command syntax) and the number of failures ($n-x$) are both >5 , for both populations.

The logic behind the test is as follows: we want to test the hypothesis that the proportions are equal (the null hypothesis). To do this, we assume that this "null hypothesis" is true, and calculate the probability that the differences between the two proportions occurred, under this assumption. If this probability is sufficiently low (usually, 5% is the cutoff point), we conclude that since it's so unlikely that the data could have occurred under the null hypothesis, the null hypothesis must be false, and therefore the proportions are not equal. If, on the other hand, the probability is not too low, we conclude that the data may well have occurred under the null hypothesis, and therefore there's no reason to reject it.

Commonly used notation has the letters π_1 and π_2 being used for the true population proportions (making the null hypothesis be $\pi_1=\pi_2$). TI must have been afraid that this would be confused with the real number π , so on the calculator, "p1" and "p2" are used everywhere instead.

In addition to the null hypothesis, we must have an alternative hypothesis as well - usually this is simply that the proportions are not equal. However, in certain cases, our alternative hypothesis may be that one proportion is greater or less than the other.

The arguments to 1-PropZTest(are as follows:

- x_1 - the success count in the first sample
- n_1 - the total size of the first sample (so the sample proportion would be x_1/n_1)
- x_2 - the success count in the second sample
- n_2 - the total size of the second sample (so the sample proportion would be x_2/n_2)
- *alternative* (optional if you don't include *draw?*) - determines the alternative hypothesis
 - 0 (default value) - $p_1 \neq p_2$
 - -1 (or any negative value) - $p_1 < p_2$
 - 1 (or any positive value) - $p_1 > p_2$

```
2-PropZTest
P1≠P2
z=-1.913340087
P=.055704393
P1=.45
P2=.55
↓P=.5409090909
```

Command Summary

Performs a z-test to compare two proportions.

Command Syntax

`2-PropZTest($x_1, n_1, x_2, n_2, \text{//draw?}$
//`

Menu Location

While editing a program, press:

1. STAT to access the statistics menu
2. LEFT to access the TESTS submenu
3. 6 to select 2-PropZTest, or use arrows

(outside the program editor, this will select the 2-PropZTest... interactive solver)

Calculator Compatibility

TI-83/84/+SE

Token Size

2 bytes

- *draw?* (optional) set this to 1 if you want a graphical rather than numeric result

Although you can access the 1-PropZTest command on the home screen, via the catalog, there's no need: the 1-PropZTest... interactive solver, found in the statistics menu, is much more intuitive to use - you don't have to memorize the syntax.

In either case, it's important to understand the output of 1-PropZTest. Here are the meanings of each line:

- The first line, involving p_1 and p_2 , is the alternative hypothesis.
- z is the test statistic. If the null hypothesis is true, it should be close to 0.
- p is the probability that the difference between the two proportions would occur if the null hypothesis is true. When the value is sufficiently small, we reject the null hypothesis and conclude that the alternative hypothesis is true. You should have a cutoff value ready, such as 5% or 1%. If p is lower, you "reject the null hypothesis on a 5% (or 1%) level" in technical terms.
- \hat{p}_1 is the sample proportion x_1/n_1 .
- \hat{p}_2 is the sample proportion x_2/n_2 .
- \hat{p} is the total sample proportion
- n_1 is the first sample size.
- n_2 is the second sample size.

Advanced Uses

The final optional argument of 2-PropZTest, *draw?*, will display the results in a graphical manner if you put in "1" for it. The calculator will draw the standard normal distribution, and shade the area of the graph that corresponds to the probability p . In addition, the value of z and the value of p will be displayed. You would make your conclusions in the same way as for the regular output.

Optimization

Some of the arguments of the 2-PropZTest command have default values, and the argument can be omitted if this value is used.

- The *draw?* argument can be omitted if you don't want graphical output, although you could put "0" in as well.
- If the above argument is omitted, and you're doing a two sided test, you may omit the *alternative* argument.

Example:

```
:2-PropZTest(22,50,48,100,0,0
can be
:2-PropZTest(22,50,48,100
```

Related Commands

- [1-PropZTest\(](#)
- [Z-Test\(](#)
- [2-SampZTest\(](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/2-propztest>

The 2-SampFTest Command

2-SampFTest performs an *F*-test to compare the standard deviations of two populations. This test is valid for two normally distributed populations, but is extremely sensitive to non-normality, so it should not be used unless you are certain that the populations are normal.

The logic behind the test is as follows: we want to test the hypothesis that the standard deviations of the two populations are equal (the null hypothesis). The letter σ is used for a standard deviation, so this is usually written as $\sigma_1=\sigma_2$. To do this, we assume that this "null hypothesis" is true, and calculate the probability that the difference between the two standard deviations occurred, under this assumption. If this probability is sufficiently low (usually, 5% is the cutoff point), we conclude that since it's so unlikely that the data could have occurred under the null hypothesis, the null hypothesis must be false, and therefore the deviations are not equal. If, on the other hand, the probability is not too low, we conclude that the data may well have occurred under the null hypothesis, and therefore there's no reason to reject it.

In addition to the null hypothesis, we must have an alternative hypothesis as well - usually this is simply that the two standard deviations are not equal. However, in certain cases when we have reason to suspect that one deviation is greater than the other (such as when we are trying to verify a claim that one standard deviation is greater), our alternative hypothesis may be that the first standard deviation is greater than the second ($\sigma_1>\sigma_2$) or less ($\sigma_1<\sigma_2$).

As for the 2-SampFTest command itself, there are two ways of calling it: you may give it a list of all the sample data, or the necessary statistics about the list (s_1 and s_2 the sample standard deviations, and n_1 and n_2 the sample sizes). In either case, you can indicate what the alternate hypothesis is, by a value of 0, -1, or 1 for the *alternative* argument. 0 indicates a two-sided hypothesis of $\sigma_1\neq\sigma_2$, -1 indicates $\sigma_1<\sigma_2$, and 1 indicates $\sigma_1>\sigma_2$. (In fact, the calculator will treat any negative value as -1, and any positive value as 1).

Although you can access the 2-SampFTest command on the home screen, via the catalog, there's no need: the 2-SampFTest... interactive solver, found in the statistics menu, is much more

```
2-SampFTest  
σ₁≠σ₂  
F=.25  
P=2.9476684E-4  
Sx₁=1  
Sx₂=2  
n₁=50
```

Command Summary

Performs a *F*-test to compare the standard deviations of two populations.

Command Syntax

2-SampFTest [*list1, list2, frequency1, frequency2, alternative, draw?*]
(data list input)

2-SampFTest s_1, n_1, s_2, n_2 ,
[*alternative, draw?*]
(summary stats input)

Menu Location

While editing a program, press:

1. STAT to access the statistics menu
2. LEFT to access the TESTS submenu
3. ALPHA D to select 2-SampFTest, or use arrows

Outside the program editor, this will select the 2-SampFTest... interactive solver.

Change the last keypress to ALPHA E on a TI-84+/SE with OS 2.30 or higher.

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

intuitive to use - you don't have to memorize the syntax.

In either case, it's important to understand the output of 2-SampFTest. Here are the meanings of each line:

- The first line, involving σ_1 and σ_2 , is the alternative hypothesis.
- F is the test statistic, the ratio of the standard deviations. If the null hypothesis is true, it should be close to 1.
- p is the probability that the difference between σ_1 and σ_2 (the two standard deviations) would occur if the null hypothesis is true. When the value is sufficiently small, we reject the null hypothesis and conclude that the alternative hypothesis is true. You should have a cutoff value ready, such as 5% or 1%. If p is lower, you "reject the null hypothesis on a 5% (or 1%) level" in technical terms.
- Sx_1 and Sx_2 are the two sample standard deviations.
- $x\bar{1}$ and $x\bar{2}$ are the two sample means. They aren't used in the calculation, and will only be shown with the data list syntax.
- n_1 and n_2 are the sample sizes.

Advanced Uses

The final optional argument of 2-SampFTest, *draw?*, will display the results in a graphical manner if you put in "1" for it. The calculator will draw the distribution, and shade the area of the graph that corresponds to the probability p. In addition, the value of F and the value of p will be displayed. You would make your conclusions in the same way as for the regular output.

As with most other statistical commands, you may use frequency lists in your input (when using the data list syntax). If you do, then both lists must have frequencies, and the order of the arguments would be *list1, list2, frequency1, frequency2*.

Optimization

Some of the arguments of the 2-SampFTest command have default values, and the argument can be omitted if this value is accepted.

- The *draw?* argument can be omitted if you don't want graphical output, although you could put "0" in as well.
- If the above argument is omitted, and you're doing a two sided test, you may omit the *alternative* argument.
- With data list input, you can always omit the frequency lists if you won't be using them.
- With data list input, if the flags that go at the end are omitted, and you're using the default lists L1 and L2, you may omit those as well.

Example:

```
: 2-SampFTest L1, L2, 0  
can be  
: 2-SampFTest
```

Related Commands

- [Z-Test](#)
- [T-Test](#)

The 2-SampTInt Command

The 2-SampTInt command uses the techniques of T Intervals to compute an interval for the **difference** between the means of two independent populations, at a specified confidence level. Use 2-SampTInt(when you have two independent variables to compare, and you don't know their standard deviations. The 2-SampTInt command assumes that both your variables are normally distributed, but it will work for other distributions if the sample size is large enough.

There are two ways to call this command: by supplying it with needed sample statistics (mean, standard deviation, and sample size, for both data sets), or by entering two lists and letting the calculator work the statistics out. In either case, you will need to enter the desired confidence level as well.

In the summary stats syntax, x_1 and x_2 the two sample means, s_1 and s_2 are the two sample standard deviations, and n_1 and n_2 the two sample sizes.

The output will contain an open interval (a, b) that is your answer: the difference between the two means will lie in this interval. Specifically, it is the second mean subtracted from the first - $\mu_1 - \mu_2$. If you're interested in the reverse difference, just flip the signs on the interval.

Tip: don't use this command in a matched-pairs setting when you can match the two samples up by units or subjects. Instead, take the difference between the two samples in each matched pair, and use a regular TInterval.

Sample Problem

You want to compare the average height of a freshman and a senior at your school. You haven't asked everyone, but you took a random sample of 40 people from each class and found out their heights (and stored them to L₁ and L₂). You've decided to use a 95% confidence interval.

Based on the data list syntax for a 2-SampTInt, here is your code:

```
2-SampTInt  
(-11.44, -8.558)  
df=207.9361544  
x1=60  
x2=70  
Sx1=5  
Sx2=5.6
```

Command Summary

Using either already-calculated statistics, or two data sets, computes a T confidence interval for the difference between two sample means.

Command Syntax

2-SampTInt *list1, list2, [frequency1], [frequency2], [confidence level, pooled]*
(data list input)

2-SampTInt $x_1, s_1, n_1, x_2, s_2, n_2,$
[confidence level, pooled]
(summary stats input)

Menu Location

When editing a program, press:

1. STAT to access the statistics menu
2. LEFT to access the TESTS submenu
3. 0 to select 2-SampTInt, or use arrows

(this key sequence will give you the 2-SampTInt... screen outside a program)

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

```
:2-SampTInt L1,L2,95  
you can also use  
:2-SampTInt L1,L2,.95
```

Alternatively, you could calculate the mean and sample size and enter those instead. The sample size in this case is 40 for both data sets; let's say the means were 57 inches and 67 inches and the standard deviations 5.2 and 7.1 inches. You now have all the needed statistics:

- x_1 is the mean height of freshmen: 57 inches
- s_1 is the sample standard deviation for freshmen: 5.2 inches
- n_1 is the number of freshmen in the sample: 40
- x_2 is the mean height of seniors: 67 inches
- s_2 is the sample standard deviation for seniors: 7.1 inches
- n_2 is the number of seniors in the sample: 40

This means that the code is:

```
:2-SampTInt 57,5.2,40,67,7.1,40,95  
you can also use  
:2-SampTInt 57,5.2,40,67,7.1,40,.95
```

Of course, the main use of the 2-SampTInt command is in a program. While you can enter the command on the home screen as well (just look in the catalog for it), it would probably be easier to select 2-SampTInt... from the STAT>TEST menu (see the sidebar), since you don't have to remember the syntax.

Advanced Uses

As with most other statistical commands, you can add frequencies to the lists (only with the data list syntax, of course); if you do, both lists must have frequencies, and the arguments go in the order *first data list, second data list, first freq. list, second freq. list*. Each frequency list must contain non-negative real numbers, which can't be all 0.

There is a final argument to 2-SampTInt: *pooled*. It can be either 0 or 1 (although any argument that isn't 0 will get treated as a 1); the default value is 0. If the value is 1, then the variances will be pooled: that is, the calculator will assume that the variances of the two populations are equal, and use a combined form of the two standard deviations in place of each population's individual standard deviation. Set this flag if you have reason to believe that the standard deviations are equal.

Optimization

Using the data list syntax, all items are optional: the calculator will assume you want to use L1 and L2 for your data unless other lists are supplied, and that the confidence level you want is 95% unless you give another one. Using the summary stats syntax, the confidence level is also optional - again, the calculator will assume 95%. This means we can rewrite our code above in a simpler manner:

```
:2-SampTInt L1,L2,95  
can be  
:2-SampTInt
```

```
:2-SampTInt 57,5.2,40,67,7.1,40,95  
can be  
:2-SampTInt 57,5.2,40,67,7.1,40
```

Related Commands

- TInterval
- ZInterval
- 2-SampZInt

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/2-samptint>

The 2-SampTTest Command

2-SampTTest performs a t significance test to compare the means of two populations. This test is valid for simple random samples from populations with unknown standard deviations. In addition, either the populations must be normally distributed, or the sample sizes have to be sufficiently large (usually, greater than 10).

The logic behind the test is as follows: we want to test the hypothesis that the true means of the two populations are equal (the null hypothesis). The letter μ is used for a population mean, so this is usually written as $\mu_1=\mu_2$. To do this, we assume that this "null hypothesis" is true, and calculate the probability that the difference between the two means occurred, under this assumption. If this probability is sufficiently low (usually, 5% is the cutoff point), we conclude that since it's so unlikely that the data could have occurred under the null hypothesis, the null hypothesis must be false, and therefore the means are not equal. If, on the other hand, the probability is not too low, we conclude that the data may well have occurred under the null hypothesis, and therefore there's no reason to reject it.

In addition to the null hypothesis, we must have an alternative hypothesis as well - usually this is simply that the two means are not equal. However, in certain cases when we have reason to suspect that one mean is greater than the other (such as when we are trying to verify a claim that one mean is greater), our alternative hypothesis may be that the first mean is greater than the second ($\mu_1>\mu_2$) or less ($\mu_1<\mu_2$).

As for the 2-SampTTest command itself, there are

```
2-SampTTest  
 $\mu_1 \neq \mu_2$   
 $t = -13.16162401$   
 $P = 3.95922E-29$   
 $df = 206.4771307$   
 $\bar{x}_1 = 60$   
 $\downarrow \bar{x}_2 = 70$ 
```

Command Summary

Performs a t significance test to compare the means of two populations.

Command Syntax

```
2-SampTTest [list1, list2,  
frequency1, frequency2,  
alternative, pooled?, draw?]  
(data list input)
```

```
2-SampTTest x1, s1, n1, x2, s2, n2,  
[alternative, pooled?, draw?]  
(summary stats input)
```

Menu Location

While editing a program, press:

1. STAT to access the statistics menu
2. LEFT to access the TESTS submenu
3. 4 to select 2-SampTTest, or use arrows

(outside the program editor, this will

two ways of calling it: you may give it a list of all the sample data, or the necessary statistics about the list (x_1 and x_2 are the sample means, s_1 and s_2 the sample standard deviations, and n_1 and n_2 the sample sizes). In either case, you can indicate what the alternate hypothesis is, by a value of 0, -1, or 1 for the *alternative* argument. 0 indicates a two-sided hypothesis of $\mu_1 \neq \mu_2$, -1 indicates $\mu_1 < \mu_2$, and 1 indicates $\mu_1 > \mu_2$. (In fact, the calculator will treat any negative value as -1, and any positive value as 1).

(outside the program editor, this will select the 2-SampTTest... interactive solver)

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

Although you can access the 2-SampTTest

command on the home screen, via the catalog, there's no need: the 2-SampTTest... interactive solver, found in the statistics menu, is much more intuitive to use - you don't have to memorize the syntax.

In either case, it's important to understand the output of 2-SampTTest. Here are the meanings of each line:

- The first line, involving μ_1 and μ_2 , is the alternative hypothesis.
- t is the test statistic, the standardized difference between the means. If the null hypothesis is true, it should be close to 0.
- p is the probability that the difference between μ_1 and μ_2 (the two means) would occur if the null hypothesis is true. When the value is sufficiently small, we reject the null hypothesis and conclude that the alternative hypothesis is true. You should have a cutoff value ready, such as 5% or 1%. If p is lower, you "reject the null hypothesis on a 5% (or 1%) level" in technical terms.
- $x_{\bar{1}}$ and $x_{\bar{2}}$ are the two sample means.
- Sx_1 and Sx_2 are the two sample standard deviations.
- n_1 and n_2 are the sample sizes.

Sample Problem

Your school claims that the average SAT score of students at the school is higher than at a rival school. You took samples of SAT scores from students at both schools (and stored them to L1 and L2).

Since the school's claim is that your school's score is higher, that will be your alternative hypothesis ($\mu_1 > \mu_2$), which corresponds to a value of 1. The code you'd use is:

```
: 2-SampTTest L1,L2,1
```

Alternatively, you could calculate the mean, standard deviation, and size of your samples, and put those into the command instead. Suppose you obtained SAT scores from 60 students at your school and 40 students at the rival school, the means were 1737 and 1623, and the standard deviation 211 and 218. Then your code is:

```
: 2-SampTTest 1737,211,60,1623,218,40,1
```

You will see the following output:

```
2-SampTTest
```

```
 $\mu_1 > \mu_2$ 
z=2.594854858
p=.0056059824
x1=1737
x2=1623
Sx1=211
Sx2=218
n1=60
n2=40
```

The most important part of this output is "p=.0056059824". This value of p is smaller than 1% or 0.01. This is significant on the 1% level, so we reject the null hypothesis and conclude that the alternative hypothesis is true: $\mu_1 > \mu_2$, that is, your school's average SAT score is indeed higher.

Advanced Uses

The final optional argument of 2-SampTTest, *draw?*, will display the results in a graphical manner if you put in "1" for it. The calculator will draw the distribution, and shade the area of the graph beyond the t statistic. In addition, the value of t and the value of p will be displayed (the value of p corresponds to the shaded area). You would make your conclusions in the same way as for the regular output.

The optional argument *pooled?*, if given a nonzero value, will pool the standard deviations to find a combined value which will then be used for both populations. Use this feature if you have reason to believe that the two populations have the same standard deviation.

As with most other statistical commands, you may use a frequency list in your input (when using the data list syntax). If you do, then both lists must have frequencies, and the order of the arguments would be *list1, list2, frequency1, frequency2*.

Optimization

Some of the arguments of the 2-SampTTest command have default values, and the argument can be omitted if this value is accepted.

- The *draw?* argument can be omitted if you don't want graphical output, although you could put "0" in as well.
- If the *draw?* argument is omitted, you can omit the *pooled?* argument if you do not want your standard deviations pooled.
- If both the above arguments are omitted, and you're doing a two sided test, you may omit the *alternative* argument.
- With data list input, you can always omit the frequency lists if you won't be using them.
- With data list input, if the flags that go at the end are omitted, and you're using the default lists L1 and L2, you may omit those as well.

The code in the sample problem above can't be optimized, because the *alternative* argument is 1:

```
: 2-SampTTest L1, L2, 1
```

However, if we were doing a two-sided test, we could omit the *alternative* argument as well as the lists:

```
:2-SampTTest L1,L2,0  
can be just  
:2-SampTTest
```

Related Commands

- [T-Test](#)
- [Z-Test\(](#)
- [2-SampZTest\(](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/2-sampttest>

The 2-SampZInt(Command

The 2-SampZInt(command uses the techniques of Z Intervals to compute an interval for the **difference** between the means of two independent populations, at a specified confidence level. Use 2-SampZInt(when you have two independent variables to compare, and you already know their standard deviations. The 2-SampZInt(command assumes that both variables are distributed normally, but it will work for other distributions if the sample size is large enough.

There are two ways to call this command: by supplying it with needed sample statistics (mean and sample size, for both data sets), or by entering two lists and letting the calculator work the statistics out. In either case, you will need to enter the standard deviation and desired confidence level as well.

In the data list syntax, σ_1 and σ_2 are the two standard deviations.

In the summary stats syntax, σ_1 and σ_2 are the two standard deviations, x_1 and x_2 the two sample means, and n_1 and n_2 the two sample sizes.

The output will contain an open interval (a, b) that is your answer: the difference between the two means will lie in this interval. Specifically, it is the second mean subtracted from the first - $\mu_1 - \mu_2$. If you're interested in the reverse difference, just flip the signs on the interval.

Tip: don't use this command in a matched-pairs setting when you can match the two samples up by units or subjects. Instead, take the difference between the two samples in each matched pair, and use a regular ZInterval.

```
2-SampZInt  
( $-1.036$ ,  $-.4642$ )  
 $\bar{x}_1=0$   
 $\bar{x}_2=.75$   
 $n_1=101$   
 $n_2=88$ 
```

Command Summary

Using either already-calculated statistics, or two data sets, computes a Z confidence interval for the difference between two sample means.

Command Syntax

2-SampZInt(σ_1 , σ_2 , [*list1, list2, frequency1, frequency2, confidence level*])
(data list input)

2-SampZInt(σ_1 , σ_2 , x_1 , n_1 , x_2 , n_2 , level)
(summary stats input)

Menu Location

When editing a program, press:

1. STAT to access the statistics menu
2. LEFT to access the TESTS submenu
3. 9 to select 2-SampZInt(, or use arrows

Sample Problem

You want to compare the average height of a freshman and a senior at your school. You haven't asked everyone, but you took a random sample of 40 people from each class and found out their heights (and stored them to L₁ and L₂). You've read in your textbook that the standard deviation of teenagers' heights is usually 6 inches. You've decided to use a 95% confidence interval.

Based on the data list syntax for a 2-SampZInt(), here is your code:

```
:2-SampZInt(6,6,L1,L2,95
you can also use
:2-SampZInt(6,6,L1,L2,.95)
```

Alternatively, you could calculate the mean and sample size and enter those instead. The sample size in this case is 40 for both data sets; let's say the means were 57 inches and 67 inches. You now have all the needed statistics:

- σ_1 is the standard deviation for freshmen: 6 inches
- σ_2 is the standard deviation for seniors: also 6 inches
- x_1 is the mean height of freshmen: 57 inches
- n_1 is the number of freshmen in the sample: 40
- x_2 is the mean height of seniors: 67 inches
- n_2 is the number of seniors in the sample: 40

This means that the code is:

```
:2-SampZInt(6,6,57,40,67,40,95
you can also use
:2-SampZInt(6,6,57,40,67,40,.95)
```

Of course, the main use of the 2-SampZInt() command is in a program. While you can enter the command on the home screen as well (just look in the catalog for it), it would probably be easier to select 2-SampZInt... from the STAT>TEST menu (see the sidebar), since you don't have to remember the syntax.

Advanced Uses

As with most other statistical commands, you can add frequencies to the lists (only with the data list syntax, of course); if you do, both lists must have frequencies, and the arguments go in the order *first data list, second data list, first freq. list, second freq. list*. Each frequency list must contain non-negative real numbers, which can't be all 0.

Optimization

Using the data list syntax, all items but the standard deviations are optional: the calculator will assume you want to use L₁ and L₂ for your data unless other lists are supplied, and that the

(this key sequence will give you the 2-SampZInt... screen outside a program)

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

confidence level you want is 95% unless you give another one. Using the summary stats syntax, the confidence level is also optional - again, the calculator will assume 95%. This means we can rewrite our code above in a simpler manner:

```
:2-SampZInt(6,6,L1,L2,95  
can be  
:2-SampZInt(6,6
```

```
:2-SampZInt(6,6,57,40,67,40,95  
can be  
:2-SampZInt(6,6,57,40,67,40
```

Related Commands

- [ZInterval](#)
- [TInterval](#)
- [2-SampTInt](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/2-sampzint>

The 2-SampZTest(Command

2-SampZTest(performs a z significance test to compare the means of two populations. This test is valid for simple random samples from populations with known standard deviations. In addition, either the populations must be normally distributed, or the sample sizes have to be sufficiently large (usually, greater than 10).

The logic behind the test is as follows: we want to test the hypothesis that the true means of the two populations are equal (the null hypothesis). The letter μ is used for a population mean, so this is usually written as $\mu_1=\mu_2$. To do this, we assume that this "null hypothesis" is true, and calculate the probability that the difference between the two means occurred, under this assumption. If this probability is sufficiently low (usually, 5% is the cutoff point), we conclude that since it's so unlikely that the data could have occurred under the null hypothesis, the null hypothesis must be false, and therefore the means are not equal. If, on the other hand, the probability is not too low, we conclude that the data may well have occurred under the null hypothesis, and therefore there's no reason to reject it.

In addition to the null hypothesis, we must have an alternative hypothesis as well - usually this is simply that the two means are not equal. However, in

```
2-SampZTest  
 $\mu_1 \neq \mu_2$   
 $z = -14.31356171$   
 $P = 1.881711 \times 10^{-46}$   
 $\bar{x}_1 = 50$   
 $\bar{x}_2 = 52$   
 $n_1 = 100$ 
```

Command Summary

Performs a z significance test to compare the means of two populations.

Command Syntax

```
2-SampZTest( $\sigma_1$ ,  $\sigma_2$  //list2//,  
//frequency1//, //frequency2//,  
//alternative//, //draw?//  
(data list input)
```

```
2-SampZTest( $\sigma_1$ ,  $\sigma_2$   $x_1$ ,  $n_1$ ,  $x_2$ ,  $n_2$ ,  
//draw?//  
(summary stats input)
```

Menu Location

While editing a program, press:

certain cases when we have reason to suspect that one mean is greater than the other (such as when we are trying to verify a claim that one mean is greater), our alternative hypothesis may be that the first mean is greater than the second ($\mu_1 > \mu_2$) or less ($\mu_1 < \mu_2$).

As for the 2-SampZTest(command itself, there are two ways of calling it: after giving the two standard deviations, you may give it a list of all the sample data, or the necessary statistics about the list (x_1 and x_2 are the sample means, and n_1 and n_2 are the sample sizes). In either case, you can indicate what the alternate hypothesis is, by a value of 0, -1, or 1 for the *alternative* argument. 0 indicates a two-sided hypothesis of $\mu_1 \neq \mu_2$, -1 indicates $\mu_1 < \mu_2$, and 1 indicates $\mu_1 > \mu_2$. (In fact, the calculator will treat any negative value as -1, and any positive value as 1).

Although you can access the 2-SampZTest(command on the home screen, via the catalog, there's no need: the 2-SampZTest... interactive solver, found in the statistics menu, is much more intuitive to use - you don't have to memorize the syntax.

In either case, it's important to understand the output of 2-SampZTest. Here are the meanings of each line:

- The first line, involving μ_1 and μ_2 , is the alternative hypothesis.
- z is the test statistic, the standardized difference between the means. If the null hypothesis is true, it should be close to 0.
- p is the probability that the difference between μ_1 and μ_2 (the two means) would occur if the null hypothesis is true. When the value is sufficiently small, we reject the null hypothesis and conclude that the alternative hypothesis is true. You should have a cutoff value ready, such as 5% or 1%. If p is lower, you "reject the null hypothesis on a 5% (or 1%) level" in technical terms.
- $x\bar{}$ and $x\bar{}$ are the two sample means.
- n_1 and n_2 are the sample sizes.

Sample Problem

Your school claims that the average SAT score of students at the school is higher than at a rival school. You took samples of SAT scores from students at both schools (and stored them to L1 and L2). Although you didn't know the standard deviations, you decided to use the value 200 that you found online as an estimate.

You now have all the data. You're assuming σ_1 and σ_2 are both 200; the two data lists are L1 and L2. Since the school's claim is that your school's score is higher, that will be your alternative hypothesis ($\mu_1 > \mu_2$), which corresponds to a value of 1. The code you'd use is:

```
: 2-SampZTest(200,200,L1,L2,1)
```

Alternatively, you could calculate the mean and sample size of your sample, and put those into the command instead. Suppose you obtained SAT scores from 60 students at your school and 40 students at the rival school, and that the means were 1737 and 1623. Then your code is:

1. STAT to access the statistics menu
2. LEFT to access the TESTS submenu
3. 3 to select 2-SampZTest(, or use arrows

(outside the program editor, this will select the 2-SampZTest... interactive solver)

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

```
: 2-SampZTest(200,200,1737,60,1623,40,1
```

You will see the following output:

```
Z-Test  
μ1>μ2  
z=2.792418307  
p=.0026158434  
x1=1737  
x2=1623  
n1=60  
n2=40
```

The most important part of this output is "p=.0026158434". This value of p is much smaller than 1% or 0.01. This is significant on the 1% level, so we reject the null hypothesis and conclude that the alternative hypothesis is true: $\mu_1 > \mu_2$, that is, your school's average SAT score is indeed higher.

Advanced Uses

The final argument of 2-SampZTest(, *draw?*, will display the results in a graphical manner if you put in "1" for it. The calculator will draw the **standard** normal curve, and shade the area of the graph beyond the z statistic. In addition, the value of z and the value of p will be displayed (the value of p corresponds to the shaded area). You would make your conclusions in the same way as for the regular output.

As with most other statistical commands, you may use a frequency list in your input (when using the data list syntax). If you do, then both lists must have frequencies, and the order of the arguments would be *list1, list2, frequency1, frequency2*.

Optimization

Most of the arguments of the 2-SampZTest(command have default values, and the argument can be omitted if this value is accepted.

- The *draw?* argument can be omitted if you don't want graphical output, although you could put "0" in as well.
- If the *draw?* argument is omitted, you can omit the *alternative* argument to use a two-sided test ($\mu_1 \neq \mu_2$). If you include the *draw?* argument, you have to include this - otherwise there will be confusion as to what the 5th argument means.
- With data list input, you can always omit the frequency lists if you won't be using them.
- With data list input, if the *draw?* and *alternative* arguments are omitted, and your data is in L1 and L2 (and you're not using frequency lists), you may omit L1 and L2 - those are default parameters. However, if *alternative* or *draw?* is present, you have to include it, or else the syntax may be confused with the syntax for summary stats input.

The code in the sample problem above can't be optimized, because the *alternative* argument is 1:

```
: 2-SampZTest(200,200,L1,L2,1
```

However, if we were doing a two-sided test, we could omit the *alternative* argument as well as the lists:

```
: 2-SampZTest(200,200,L1,L2,0  
can be  
: 2-SampZTest(200,200
```

Related Commands

- Z-Test
- T-Test
- 2-SampTTest

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/2-sampztest>

The 2-Var Stats Command

This command calculates a bunch of common (and a few uncommon) statistics for a pair of lists (it uses L1 and L2 by default, but you can use any list by supplying it as an argument). You have to store the lists to variables first, though, before calculating statistics for them. For example:

```
: {5,12,7,8,4,9→L1  
: {1,0,2,5,7,4→L2  
: 2-Var Stats
```

The calculator treats the two lists as a list of ordered pairs. Some of the statistics calculated assume that this is the case, and the two lists are the same size: an error will occur if the lists don't match.

Like other statistical commands, you can use a frequency list as well, for cases where one element occurs more times than another (you can do this with a normal list, too, but that might be inconvenient when an element occurs very many times). There is only one frequency list for both data lists, and the frequency applies to the ordered pair formed by an element taken from each list. For example:

```
: {1,2,3→L1  
: {1,2,3→L2  
: {5,3,2→L3  
: 2-Var Stats L1,L2,L3
```

is the frequency-list equivalent of:

```
2-Var Stats  
x̄=3  
Σx=15  
Σx²=55  
Sx=1.58113883  
σx=1.414213562  
n=5
```

Command Summary

Calculates some common statistics for two lists of data, and stores them to statistical variables. They're also displayed in a scrollable list, if done outside a program.

Command Syntax

2-Var Stats [*list1*, *list2*, [*freqlist*]]

Menu Location

Press:

1. STAT to access the statistics menu
2. LEFT to access the CALC submenu
3. 2 to select 2-Var Stats, or use arrows

Calculator Compatibility

TI-83/84/+SE

```
:{1,1,1,1,1,2,2,2,3,3→L1  
:{1,1,1,1,1,2,2,2,3,3→L2  
:2-Var Stats
```

Token Size

1 byte

When you're running it from the home screen, 2-Var Stats will display the statistics; this won't happen if you do it inside a program. Either way, it will also store what it calculated to the statistics variables found in VARS>Statistics... The variables 2-Var Stats affects are:

- \bar{x} is the mean (average) of the first list
- Σx is the sum of the first list
- Σx^2 is the sum of the squares of the first list
- S_x is the sample standard deviation of the first list
- σ_x is population standard deviation of the first list
- $\min X$ is the minimum element of the first list
- $\max X$ is the maximum element of the first list
- \bar{y} is the mean (average) of the second list
- Σy is the sum of the second list
- Σy^2 is the sum of the squares of the second list
- S_y is the sample standard deviation of the second list
- σ_y is population standard deviation of the second list
- $\min Y$ is the minimum element of the second list
- $\max Y$ is the maximum element of the second list
- Σxy is the sum of products of each matching pair of elements in the lists
- n is the number of elements in both lists

2-Var Stats will not work with "reserved" list names that the calculator uses internally. The only known such reserved list is the list RESID, and there's no reason to suspect there are any others. Ans, TblInput, and any expression which resolves to a list, are also not appropriate for this command: store all of these to a list before doing 2-Var Stats on them.

Advanced uses

If you consider the two lists to be vectors, then Σxy is their dot product, and Σx^2 and Σy^2 are the squares of their norms; math done with these and other statistics can produce the shortest (but not necessarily quickest) way to calculate many vector operations.

Optimization

Aside from statistical analysis, 2-Var Stats can also be used when you want to use the values it calculates more than once. This will save on size, since, for example Σx takes up less space than $\text{sum}(L1)$, but considering how many calculations 2-Var Stats makes, it will usually be slower.

Related Commands

- [1-Var Stats](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/2-var-stats>

The ³ Command

The 3 command raises an input to the third power. It has exactly the same function as " 3 ", but is one byte smaller. If used on a list, it will return a list with all of the elements cubed. If used on a matrix, it will return the third matrix power of the input matrix.

```
23
8
{1, -2, 3}3
{1 -8 27}
[[2, -1] [-3, 0]]3
[[20 -7]
[-21 6]]
```

Advanced Uses

One trick with 3 is to use it to save space (at the cost of speed) when using hard-coded values. For instance, use 5^3 instead of 125 to save one byte.

Optimization

Use this command instead of 3 in all instances.

```
:X^3
can be
:X3
```

Related Commands

- $\frac{-1}{2}$
- $\frac{2}{2}$

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/3>

The ${}^3\sqrt{}$ Command

Takes the cube root of a positive or negative number. It works exactly the same as ${}^3\sqrt{x}$ or ${}^3(x)$ but is smaller and uses an ending parenthesis. If used on a list, it will return a list with the cube root of each element.

```
 ${}^3\sqrt{8}$ 
2
 ${}^3\sqrt{2}$ 
1.25992105
```

${}^3\sqrt{3}$	27
${}^3\sqrt{3}$	27

Command Summary

Raises the input to the third power.

Command Syntax

*value*³

Menu Location

While editing a program, press:

1. MATH to enter the MATH menu
2. 3 or use the arrow keys to select.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

${}^3\sqrt{27}$	3
${}^3\sqrt{{1,8,27,64}}$	{1 2 3 4}

Command Summary

```
 $\sqrt[3]{\{1, -8, 27\}}$   
 $\{1 \ -2 \ 3\}$ 
```

For complex numbers, the principal cube root is returned, which may be different from the cube root you'd get for the same real number:

```
 $\sqrt[3]{-8}$   
-2  
 $\sqrt[3]{-8+0i}$   
1+1.732050808i
```

Optimization

Never raise something to the one-third power explicitly; use this command instead.

```
:X^(1/3)→X  
can be  
: $\sqrt[3]{X}$ 
```

Related Commands

- \wedge
- \sqrt{x}
- $\sqrt{}$

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/cube-root>

The a+bi Command

The $a+bi$ command puts the calculator into rectangular complex number mode. This means that:

- Taking square roots of negative numbers, and similar operations, no longer returns an error.
- Complex results are displayed in the form $a+bi$ (hence the name of the command)

This is the standard way of displaying complex numbers, though they can also be displayed in polar form (see $re^{\theta}i$ for more details). To extract the coefficients a and b , use the real(and imag(commands.

Advanced Uses

Take the cube root of a number.

Command Syntax

$\sqrt[3]{input}$

Menu Location

While editing a program, press:

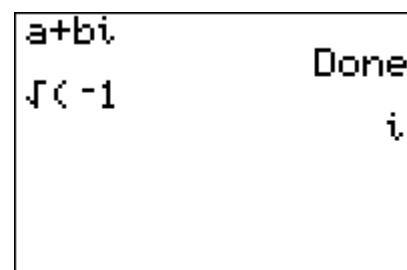
1. MATH to open the math menu
2. 4 or use the arrow keys to select.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte



Command Summary

Puts the calculator into $a+bi$ mode.

Command Syntax

$a+bi$

Rather than switch to a+bi mode, you might want to force the calculations to use complex numbers by making the original argument complex. The general way to do this is by adding +0i to the number. However, there may be an optimization in any particular case. See the [quadratic formula](#) routine for a good example of this.

```
Real
      Done
 $\sqrt{(-1)}$       (causes an error)
 $\sqrt{(-1+0i)}$     i
```

Menu Location

Press:

1. MODE to access the mode menu.
2. Use the arrow keys and ENTER to select a+bi

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

Related Commands

- [Real](#)
- [re^θi](#)

See Also

- [Quadratic Formula](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/a-bi>

The abs(Command

abs(x) returns the absolute value of the real number x. Also works on a list or matrix of real numbers.

```
abs(3)
      3
abs(-3)
      3
```

For complex numbers, abs(z) returns the absolute value (also known as the complex modulus, norm, or a hundred other terms) of the complex number z. If z is represented as x+iy where x and y are both real, abs(z) returns $\sqrt{x^2+y^2}$. Also works on a list of complex numbers.

```
abs(3+4i)
      5
```

abs(12)	12
abs(-12)	12
abs(3-4i)	5

Command Summary

Returns the absolute value of a real number, and the complex absolute value of a complex number.

Command Syntax

abs(value)

Menu Location

Press:

Optimization

The `abs`(command, used properly, may be a smaller method of testing if a variable is in some range. For example:

```
: If 10<X and X<20  
can be  
: If 5>abs(X-15)
```

In general, the first number, A, in the expression `A>abs(X-B)` should be half the length of the range, half of 10 in this case, and the second number, B, should be the midpoint of the range (here, 15).

This can be taken to extreme degrees. For example, the following code uses `abs`(three times to test if X is the `getKey` keycode of one of the keys 1, 2, 3, 4, 5, 6, 7, 8, or 9:

```
: If 2>abs(5-abs(5-abs(X-83
```

For complex numbers given by a separate real and complex part, `abs(X+iY)` can be optimized to `R▶Pr(X,Y)`.

Related Commands

- `angle`(
- `real`(
- `imag`(
- `conj`(
- `R▶Pr`(

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/abs>

The `and` Command

`and` takes two numbers, variables, or expressions and tests to see if they are both True (not equal to 0). If they are, it returns 1. If either input is False (0), it returns 0. Note that the order of the operators doesn't matter (i.e. `and` is commutative), and that multiple `and`'s can be used together

```
: 0 and 0  
0  
: 0 and 1
```

1. MATH to access the math menu.
2. RIGHT to access the NUM submenu.
3. ENTER to select `abs`(.

Alternatively, press:

1. MATH to access the math menu.
2. RIGHT twice to access the CPX (complex) submenu.
3. 5 to select `abs`(, or use arrows.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

<code>2+2=4</code>	1
<code>2+2=5</code>	0
<code>2+2=4 and 2+2=5</code>	0

Command Summary

Returns the logical value of `value1`

```

0
:1 and 2      (2 counts as True
1

:1→X
:X and 2+2      (you can use vari
1

:1 and 1 and 2-2      (the last input e
0

```

Optimization

Multiplying two values has the same truth value as and; thus, 'and' can sometimes be replaced by multiplication. Because the calculator does implicit multiplication, meaning it automatically recognises when you want to multiply, you don't need to use the * sign.

```

:If A and B
can be
:If AB

```

However, do not use this optimization if A and B might be expected to take on large values, as an overflow error might occur.

Related Commands

- [or](#)
- [xor](#)
- [not\(](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/and>

The angle(Command

`angle(z)` returns the [complex argument](#) (also known as the polar angle) of the complex number z . If z is represented as $x+iy$ where x and y are both real, `angle(z)` returns $R\blacktriangleright P\theta(x,y)$ (which is equivalent to $\tan^{-1}(y/x)$ if x is nonzero). Also works on a list of complex numbers.

```

angle(3+4i)
.927295218
R►Pθ(3,4)

```

Returns the logical value of `value1`, and `value2` being true.

Command Syntax

`value1 and value2`

Menu Location

Press:

1. 2nd TEST to access the test menu.
2. RIGHT to access the LOGIC submenu.
3. ENTER to select and.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

```

angle(i
1.570796327
angle(-1
3.141592654
angle(e^(2i
2

```

Command Summary

When writing a complex number z in the form $re^{i\theta}$ (or, equivalently, $r(\cos\theta + i\sin\theta)$), then θ is equal to the value of $\text{angle}(z)$, suitably reduced so that the result returned is in the interval $-\pi < \theta \leq \pi$.

The `angle(` command also works on matrices, though not in any useful way: `angle([A])` will return a matrix of the same size as `[A]`, but with all elements 0. If you plan to use this, **don't**: `0[A]` does the same thing, but is smaller and not as questionable (because this behavior is clearly unintentional on TI's part, and may be changed in an OS update).

Related Commands

- [`abs\(`](#)
- [`conj\(`](#)
- [`real\(`](#)
- [`imag\(`](#)
- [`R►Pθ\(`](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/angle>

The ANOVA(Command

The ANOVA (analysis of variance) command is used to test if there is a significant difference between the means of several populations (this is an extension of the two-sample t-test which compares only two populations). The calculator assumes the null hypothesis, that all means are equal, and returns a probability value, p , of the differences in the data occurring if the null hypothesis were true. If p is small (usually, if it's less than .05), then it's unlikely we'd get such differences just by chance if the null hypothesis were true, so we reject it and conclude that at least one of the means is different.

There are two reasons why we don't test the means in pairs using a simpler test. First of all, it would take a long time: there's so many pairs to compare. Second of all, when you're doing many tests, there's a high probability you'll get a low p -value by chance. Imagine that you're doing 10 tests. If the probability of getting a low p -value on one test is .05, then the probability that at least one test will return one is $1 - .95^{10}$: about 0.4 - this is quite likely to happen. The ANOVA test avoids this by having only one null

Returns the complex argument of a complex number.

Command Syntax

`angle(z)`

Menu Location

Press:

1. MATH to access the math menu.
2. RIGHT, RIGHT to access the CPX (complex) submenu
3. 4 to select `angle()`, or use arrows.

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

One-way ANOVA $F=27$ $P=.001$ Factor $df=2$ $SS=54$ $\downarrow MS=27$
--

Command Summary

Performs a one way ANOVA (analysis of variance) test to compare the means of multiple populations (up to 20).

Command Syntax

`ANOVA(list, list, ...)`

Menu Location

Press:

hypothesis to test.

If you're only interested in the result of the test, the only thing you'll need in the output is the second line: "p=..." This is your p-value, and determines whether you should reject the null hypothesis or not. If you need more detail, here are the meanings of the other variables:

- **F** is the test statistic. If the null hypothesis is true, it should follow Snedecor's F distribution, and Fcdf(can be used to determine the p-value.
- For both Factor and Error:
 - MS is the mean squares (SS/df). If the null hypothesis is true, Factor MS should be roughly equal to Error MS
 - SS is the sum of squares - see the TI-83+ Manual for formulas
 - df is the number of degrees of freedom - for Factor, it's the df between the categorical variables, and for Error, it's the sum of df between each variable.
- Sxp is the pooled variation.

1. STAT to access the statistics menu
2. LEFT to access the TESTS submenu
3. ALPHA F to select ANOVA(, or use arrows

Change the last keypress to ALPHA H on a TI-84+/SE with OS 2.30 or higher.

Calculator Compatibility

TI-83/84+/SE

Token Size

2 bytes

Advanced Uses

The statistics F, p, and Sxp will be stored to the appropriate variables after this test. The other six statistics do not have a normal variable associated with them. However, the two-byte tokens 0x6237 through 0x623C are, in fact, used to store the values of Factor MS, Factor SS, Factor df, Error MS, Error SS, and Error df respectively. They can't be accessed through a menu, but if you use a hex editor to paste them into your program, you will be able to use them just like any other variable.

However, be careful because the Factor and Error tokens look exactly alike (even though they refer to different variables), and can be confused. Also, there is a chance that future OS versions will change the behavior of ANOVA(, though this is unlikely, and this trick will no longer work.

Related Commands

- 2-SampTTest
- x²-Test(
- 2-SampFTest

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/anova>

The Ans Variable

The Ans variable holds the last answer that was stored in the calculator. Because Ans is stored in a special storage area built-in to the calculator, and it is extensively used by the calculator, you cannot delete it. Not only is Ans a unique, one-of-a-kind

2	2
2*Ans	4

variable, but it is also one of the most useful variables available on the calculator because it can make your programs both smaller and faster:

- Unlike other variables which have a value type hard-coded in (i.e., a string can only hold text, and lists and matrices can only hold numbers), Ans can take on whatever value you want: a real or complex, list, matrix, or string are all acceptable.
- Along with the finance variables, Ans is faster than the real, complex, list, matrix, and string variables; and subsequently, you should try to use it as much as possible.

One of the most common places to use Ans is in place of storing a value to a variable. All you need to do to use Ans is just paste the Ans variable to the location where the variable was called, and then when the expression is evaluated, the calculator will use the current value of Ans. Using the Ans variable allows you to eliminate the variable, which helps save a little or a lot of memory (depending on the type of variable and its size).

```
: 30+5A→B  
: Disp 25A, 30+5A  
can be  
: 30+5A→B  
: Disp 25A, Ans
```

Command Summary

Returns the last answer.

Command Syntax

Ans[→Variable]

Menu Location

While editing a program, press [2nd] then [(−)]

Calculator Compatibility

TI-83/84/+SE

Token Size

1 byte

The one major drawback to using Ans is that its current value is only temporary. Whenever you store a value to a variable, place an expression or string on a line by itself, or use the optional argument of the Pause command, Ans gets updated and takes on that new value. This restriction essentially limits your use of Ans to only a single variable. If you are manipulating two or more variables, it's best to just use the variables.

There are several cases in which changing the value of a variable does not modify Ans, thus preserving its current value for later use:

- storing to an equation variable
- using the DelVar command to delete a variable (i.e., set its value to zero, if it's a real variable)
- changing the value with IS>(or DS<(.
- initializing or changing the value in a For(loop.

These cases can be very useful, allowing you to use Ans to store an expression rather than create a temporary variable for it.

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/ans>

The Archive Command

The Archive command moves a variable from RAM to the archive (also known as ROM). A quick synopsis of the difference between the two:

- Data in the archive cannot be accessed, but it's protected from RAM clears (which may occur during battery removal if not done carefully); also, the archive can hold much more data.
- Data in RAM can be accessed for calculations, but it can also be deleted during a RAM clear or accidentally overwritten by another program.

Nothing happens if the variable in question is already archived.

You might want to use this command to protect data such as saved games from being accidentally deleted. It's not, in general, a good idea to archive commonly used variables, such as the real variables A-Z, since programs usually expect to be able to access these variables without problems, and won't check if they're archived.

Also, some variables cannot be archived. These include:

- The real variables R, T, X, Y, θ , and n (due to their use in graphing)
- The equation variables Y_n , X_{nT} , Y_{nT} , r_n , u , v , and w
- The stat plots Plot#
- Window, table, and zoom variables such as TblInput or Xmin
- Statistical variables and the list L RESID
- Finance variables

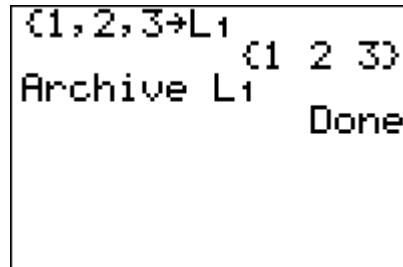
Finally, the Archive command does not work on programs when using it from a program (it does, however, archive programs from the home screen). However, an assembly program can be executed as a subroutine so that Archive and UnArchive can be used within a program. The program should however be run again afterwards.

Advanced Uses

As archived variables (and programs) can not be accessed by the calculator's inbuilt OS, archiving programs can be quite problematic when trying to execute them. However; by enabling your programs to be viewable in assembly shells, you can execute your programs without needing to unarchive them first. This is because the assembly shell copies the program to the RAM automatically, and is then executed. Closing the program will automatically remove the copy from the RAM, so no RAM is lost in the end.

Error Conditions

- ERR:ARCHIVE FULL is thrown when there isn't enough space in the archive for the variable.
- ERR:INVALID is thrown when trying to archive a program from within a program.



Command Summary

Moves a variable from RAM to the archive.

Command Syntax

Archive *variable*

Menu Location

Press:

1. 2nd MEM to access the memory menu
2. 5 to select Archive, or use arrows

Calculator Compatibility

TI-83+/84+/SE

(not available on the regular TI-83)

Token Size

2 bytes

- **ERR:VARIABLE** is thrown when trying to archive a variable that cannot be archived.

Related Commands

- [UnArchive](#)
- [DelVar](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/archive>

The Asm(Command

The Asm(command is used for running an assembly program. Unlike TI-Basic programs, assembly programs are written in the calculator's machine code directly, which makes them more powerful in both speed and functionality. However, it also means that if they crash, they crash hard — there is no built-in error menu to protect you.

Keep in mind that many assembly programs these days are written for a shell such as Ion or MirageOS. If you're dealing with one of those programs, calling Asm(on it will do nothing; you need to get the appropriate shell and run that instead.

With the AsmPrgm and AsmComp(commands, you can create small assembly programs yourself, directly on the calculator.

Error Conditions

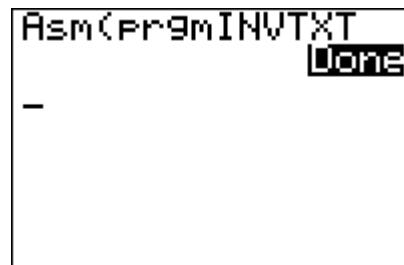
- **ERR:INVALID** is thrown if the program isn't an assembly program.

Related Commands

- [AsmPrgm](#)
- [AsmComp\(](#)

See Also

- [Assembly Shells](#)



Command Summary

Runs an assembly program.

Command Syntax

Asm(prgmNAME)

Menu Location

This command is only found in the catalog. Press:

1. 2nd CATALOG to access the command catalog.
2. DOWN six times.
3. ENTER to select Asm(.

Calculator Compatibility

TI-83+/84+/SE

(not available on the regular TI-83)

Token Size

2 bytes

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/asm-command>

The AsmComp(Command

This command is used to compress an assembly program written using AsmPrgm into a "compiled" assembly program. This will make the program about twice as small, and protect it from being edited.

To use AsmComp(), give it the uncompressed assembly program, followed by the name you want the compiled program to have. That name can't be already taken. Since it's not easy to rename a compiled assembly program, if you want to write a program called prgmGAME, you type the uncompressed code in a program with a different name (e.g. GAMEA) and then do AsmComp(prgmGAMEA,prgmGAME).

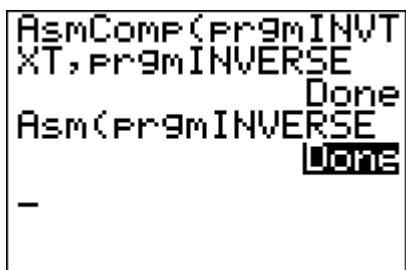
Both types of assembly programs can be run with Asm().

Error Conditions

- **ERR:DUPLICATE** is thrown if prgm*RESULT* is an already used program name;
- **ERR:INVALID** is thrown if prgm*ORIGINAL* doesn't start with AsmPrgm;
- **ERR:SYNTAX** is thrown if prgm*ORIGINAL* is not an assembly program.

Related Commands

- Asm(
- AsmPrgm



AsmComp(prgmINV
XT,prgmINVER
SE
Done
Asm(prgmINVERSE
Done
-

Command Summary

Compresses an assembly program in hexadecimal form into binary form.

Command Syntax

Asm(prgm*ORIGINAL*,prgm*RESULT*)

Menu Location

This command is only found in the catalog. Press:

1. 2nd CATALOG to access the command catalog.
2. Scroll down to AsmComp() and press enter.

Calculator Compatibility

TI-83+/84+/SE

(not available on the regular TI-83)

Token Size

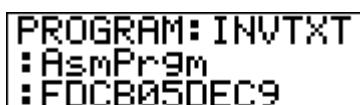
2 bytes

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/asmcomp>

The AsmPrgm Command

This command denotes the start of an assembly program in hexadecimal form. The command must go at the beginning of a program.

Using AsmPrgm is the only built-in way to create



PROGRAM: INV
XT:
AsmPrgm
FDCB05DEC9

assembly programs on the calculator, and it's not very convenient. To use it, after AsmPrgm itself, you must type in the hexadecimal values (using the numbers 0-9, and the letters A-F) of every byte of the assembly program. Even for assembly programmers, this is a complicated process: unless you've memorized the hexadecimal value of every assembly command (which is about as easy as memorizing the hexadecimal value of every TI-Basic token) you have to look every command up in a table.

In addition, it's easy to make a typo while doing this. For this reason, it's recommended **not** to use AsmPrgm to write assembly programs on the calculator, but instead write assembly programs on the computer. This also lets you use emulators and debuggers and such, as opposed to crashing your calculator (possibly permanently) every time you have a bug.

Just about the only use for AsmPrgm is to enter the hex codes for simple assembly routines that can be called from Basic programs or used for some other short task. For example, the following program will allow you to type in lowercase letters (by pressing ALPHA twice, you go into lowercase letter mode):

```
AsmPrgmFDCB24DEC9
```

To use this, create a program, and enter the code above into it. Then run the program using Asm(. Voila! Lowercase letters are now enabled.

More such short programs can be found [here](#).

Related Commands

- [Asm\(](#)
- [AsmComp\(](#)

See Also

- [Assembly Hex Codes](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/asmpgm>

The augment(Command

The augment(command is used to combine two lists or two matrices into one. For lists, this is done the

```
augment((1,2),{3}
```

Command Summary

This command must be the beginning of an assembly program.

Command Syntax

AsmPrgm

Menu Location

This command is only found in the catalog. Press:

1. 2nd CATALOG to access the command catalog.
2. Scroll down to AsmPrgm and press ENTER.

Calculator Compatibility

TI-83+/84+/SE

(not available on the regular TI-83)

Token Size

2 bytes

obvious way: adding the elements of the second on to the elements of the first. For example:

```
augment({1,2,3,4},{5,6,7}  
{1 2 3 4 5 6 7})
```

For matrices, the columns of the second matrix are added after the columns of the first matrix: an R by C matrix augmented with an R by D matrix will result in an R by (C+D) matrix. For example:

```
augment([[1][2]],[[3][4]  
[[1 3]  
[2 4]])
```

Advanced Uses

Use the T (transpose) command if you want to combine two matrices vertically, rather than horizontally. For example:

```
augment([[1,2]]T,[[3,4]]T)T  
[[1 2]  
[3 4]])
```

Optimization

You may be tempted to use `augment(` to add one element to the end of a list:

```
: augment(L1,{X→L1})
```

However, the following way is faster and more memory-efficient while the program is running (although it increases the program's size):

```
: X→L1(1+dim(L1))
```

Error Conditions

- **ERR:DATA TYPE** is thrown if you try to augment a single number to a list, a common error — use {X instead of X.
- **ERR:DIM MISMATCH** is thrown if you try to augment two matrices with a different number of rows.
- **ERR:INVALID DIM** is thrown if one of the arguments is a list with dimension 0, or if the result would have dimension over 999 (for lists) or 99x99 (for matrices).

```
,4  
{1 2 3 4}  
augment([[1][2]]  
,[[3,4][5,6]  
[[1 3 4]  
[2 5 6]])
```

Command Summary

Combines two lists or matrices into one. In the case of matrices, this is done horizontally rather than vertically.

Command Syntax

`augment(list1,list2)`

`augment(matrix1,matrix2)`

Menu Location

Press:

1. 2nd LIST to access the List menu
2. RIGHT to access the OPS submenu
3. 9 to select `augment(`, or use arrows

Alternatively, press:

1. MATRX (on the TI-83) or 2nd MATRX (TI-83+ or higher) to access the Matrix menu
2. RIGHT to access the MATH submenu
3. 7 to select `augment(`, or use arrows

Calculator Compatibility

TI-83/84/+SE

Token Size

1 byte

Related Commands

- dim(
- seq(
- T (transpose)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/augment>

The AxesOff Command

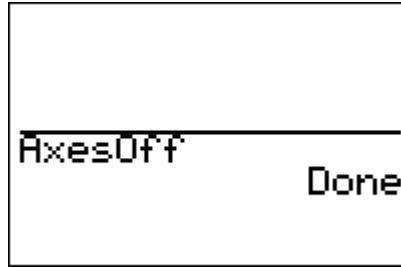
The AxesOff command disables the X and Y axes on the graph screen, so that they aren't drawn. They can be enabled again with the AxesOn command.

(the $y=x$ line that is drawn when both Seq and Web modes are enabled is also controlled by this command)

Generally, the AxesOff command should be used at the beginning of the program to disable the axes if the program is going to use the graph screen, since the axes get in the way. However, you should consider using StoreGDB and RecallGDB to save this setting if that's the case.

Related Commands

- AxesOn
- LabelOn
- LabelOff



Command Summary

Disables the X- and Y- axes on the graph screen.

Command Syntax

AxesOff

Menu Location

Press:

1. 2nd FORMAT to access the format menu.
2. Use arrows and ENTER to select AxesOff.

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/axesoff>

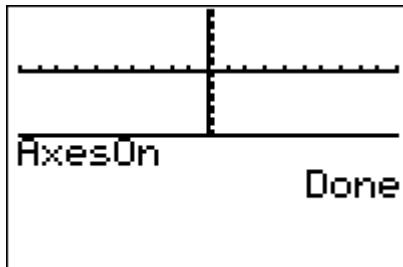
The AxesOn Command

The AxesOn command enables the X and Y axes on the graph screen, so that they are drawn. They can be disabled with the AxesOff command.

(the $y=x$ line that is drawn when both Seq and Web modes are enabled is also controlled by this command)

Related Commands

- AxesOff
- LabelOn
- LabelOff



Command Summary

Enables the X- and Y- axes on the graph screen.

Command Syntax

AxesOn

Menu Location

Press:

1. 2nd FORMAT to access the format menu.
2. Use arrows and ENTER to select AxesOn.

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/axeson>

The bal(Command

The bal(command calculates the remaining balance after n payments in an amortization schedule. It has only one required argument: n , the payment number. However, it also uses the values of the finance variables PV, PMT, and I% in its calculations.

The optional argument, *roundvalue*, is the number of digits to which the calculator will round all internal calculations. Since this rounding affects further steps, this isn't the same as using round(to round the result of bal(to the same number of digits.

Usually, you will know the values of **N**, **PV**, and **I%**, but not **PMT**. This means you'll have to use the

```
30*12→N:8/12→I%:  
100000→PV:0→FV  
0  
tvm_Pmt→PMT  
-733.7645739  
bal(180)  
76781.55951
```

Command Summary

Calculates the remaining balance after n payments in an amortization schedule.

finance solver to solve for PMT before calculating bal(); virtually always, FV will equal 0.

Sample Problem

Imagine that you have taken out a 30-year fixed-rate mortgage. The loan amount is 100000, and the annual interest rate (APR) is 8%. Payments will be made monthly. After 15 years, what amount is still left to pay?

We know the values of **N**, **I%**, and **PV**, though we still need to convert them to monthly values (since payments are made monthly). **N** is 30×12 , and **I%** is $8/12$. **PV** is just 100000.

Now, we use the finance solver to solve for PMT. Since you intend to pay out the entire loan, FV is 0. Using either the interactive TVM solver, or the `tvm_Pmt` command, we get a value of about -3.76 for PMT.

We are ready to use `bal()`. We are interested in the payment made after 15 years; this is the $15 \times 12 = 180^{\text{th}}$ payment. `bal(180)` gives us the result 781.55 — as you can see, most of the loan amount is still left to pay after 15 years.

Formulas

The calculator uses a recursive formula to calculate `bal()`:

$$\text{bal}(0) = \text{PV} \quad (1)$$

$$\text{bal}(m) = \left(1 - \frac{I\%}{100}\right) \text{bal}(m - 1) + \text{PMT} \quad (2)$$

In the case that *roundvalue* is given as an argument, the rounding is done at each step of the recurrence (which virtually forces us to use this formula). Otherwise, if no rounding is done (and assuming **I%** is not 0), we can solve the recurrence relation to get:

$$\text{bal}(m) = \frac{1 - \left(1 - \frac{I\%}{100}\right)^m}{\frac{I\%}{100}} \text{PMT} + \left(1 - \frac{I\%}{100}\right)^m \text{PV} \quad (3)$$

Error Conditions

- **ERR:DOMAIN** is thrown if the payment number is negative or a decimal.

Related Commands

- `ΣPrn(`

Command Syntax

`bal(n,[roundvalue])`

Menu Location

On the TI-83, press:

1. 2nd FINANCE to access the finance menu.
2. 9 to select `bal()`, or use arrows and ENTER.

On the TI-83+ or higher, press:

1. APPS to access the applications menu.
2. 1 or ENTER to select Finance...
3. 9 to select `bal()`, or use arrows and ENTER.

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

- Σ Int(
- tvm_Pmt

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/bal>

The binomcdf(Command

This command is used to calculate the binomial cumulative probability function. In plainer language, it solves a specific type of often-encountered probability problem, that occurs under the following conditions:

1. A specific event has only two outcomes, which we will call "success" and "failure"
2. This event is going to repeat a specific number of times, or "trials"
3. Success or failure is determined randomly with the same probability of success each time the event occurs
4. We're interested in the probability that there are **at most** N successes

For example, consider a couple that intends to have 4 children. What is the probability that at most 2 are girls?

1. The event here is a child being born. It has two outcomes "boy" or "girl". In this case, since the question is about girls, it's easier to call "girl" a success.
2. The event is going to repeat 4 times, so we have 4 trials
3. The probability of a girl being born is 50% or 1/2 each time
4. We're interested in the probability that there are at most 2 successes (2 girls)

The syntax here is `binomcdf(trials, probability, value)`. In this case:

```
:binomcdf(4,.5,2)
```

This will give .6875 when you run it, so there's a .6875 probability out of 4 children, at most 2 will be girls.

An alternate syntax for `binomcdf()` leaves off the last argument, *value*. This tells the calculator to compute a list of the results for all values. For example:

```
:binomcdf(4,.5
```

```
binomcdf(4,.5
(.0625 .3125 .6...
binomcdf(4,.5,0
.0625
binomcdf(5,0
(1 1 1 1 1 1)
```

Command Summary

Calculates the binomial cumulative probability, either at a single value or for all values

Command Syntax

for a single value:
`binomcdf(trials, probability, value)`

for a list of all values (0 to *trials*)
`binomcdf(trials, probability)`

Menu Location

Press:

1. 2ND DISTR to access the distribution menu
2. ALPHA A to select `binomcdf()`, or use arrows.

Press ALPHA B instead of ALPHA A on a TI-84+/SE with OS 2.30 or higher.

Calculator Compatibility

TI-83/84+/SE

Token Size

2 bytes

This will come to $\{.0625 .3125 .6875 .9375 1\}$ when you run it. These are all the probabilities we get when you replace "at most 2 girls" with "at most 0", "at most 1", etc. Here, .0625 is the probability of "at most 0" girls (or just 0 girls), .3125 is the probability of at most 1 girl (1 or 0 girls), etc.

Several other probability problems actually are the same as this one. For example, "less than N" girls, just means "at most N-1" girls. "At least N" girls means "at most (total-N)" boys (here we switch our definition of what a success is). "No more than", of course, means the same as "at most".

Advanced (for programmers)

The `binompdf()` and `binomcdf()` commands are the only ones apart from `seq()` that can return a list of a given length, and they do it much more quickly. It therefore makes sense, in some situations, to use these commands as substitutes for `seq()`.

Here's how to do it:

1. `cumSum(binomcdf(N,0))` gives the list $\{1 2 \dots N+1\}$, and `cumSum(not(binompdf(N,0)))` gives the list $\{0 1 2 \dots N\}$.
2. With `seq()`, you normally do math inside the list: `seq(3I^2,I,0,5)`
3. With these commands, you do the same math outside the list: `3Ans^2` where `Ans` is the list $\{0 1 \dots 5\}$.

```
:seq(2^I,I,1,5  
can be  
:cumSum(binomcdf(4,0  
:2^Ans  
which in turn can be  
:2^cumSum(binomcdf(4,0
```

In general (where `f()` is some operation or even several operations):

```
:seq(f(I),I,1,N  
can be  
:cumSum(binomcdf(N-1,0  
:f(Ans  
which can sometimes be  
:f(cumSum(binomcdf(N-1,0
```

If the lower bound on I in the `seq()` statement is 0 and not 1, you can use `binompdf()` instead:

```
:seq(f(I),I,0,N  
can be  
:cumSum(not(binompdf(N,0  
:f(Ans  
which can sometimes be  
:f(cumSum(not(binompdf(N,0
```

This will not work if some command inside `seq()` can take only a number and not a list as an

argument. For example, `seq(L1(I),I,1,5)` cannot be optimized this way.

Formulas

Since "at most N" is equivalent to "0 or 1 or 2 or 3 or ... N", and since we can combine these probabilities by adding them, we can come up with an expression for `binomcdf(` by adding up values of `binompdf(`:

$$\text{binomcdf}(n, p, k) = \sum_{i=0}^k \text{binompdf}(n, p, i) = \sum_{i=0}^k \binom{n}{i} p^i (1-p)^{n-i} \quad (1)$$

(If you're not familiar with sigma notation, $\sum_{i=0}^k$ just means "add the following up for each value of i 0 through k "

Error Conditions

- **ERR:DATATYPE** is thrown if you try to generate a list of probabilities with p equal to 0 or 1, and at least 257 trials.
- **ERR:DOMAIN** is thrown if the number of trials is at least 1 000 000, unless the other arguments make the problem trivial.

Related Commands

- `binompdf(`
- `geometpdf(`
- `geometcdf(`

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/binomcdf>

The `binompdf(` Command

This command is used to calculate the binomial probability. In plainer language, it solves a specific type of often-encountered probability problem, that occurs under the following conditions:

1. A specific event has only two outcomes, which we will call "success" and "failure"
2. This event is going to repeat a specific number of times, or "trials"
3. Success or failure is determined randomly with the same probability of success each time the event occurs
4. We're interested in the probability that there are exactly N successes

For example, consider a couple that intends to have 4 children. What is the probability that 3 of them are girls?

```
binompdf(4,.5  
.0625 .25 :375...  
binompdf(4,.5,0  
.0625  
binompdf(5,0  
{1 0 0 0 0 0}
```

Command Summary

Calculates the binomial probability, either at a single value or for all values

Command Syntax

for a single value:
`binompdf(trials probability value)`

1. The event here is a child being born. It has two outcomes "boy" or "girl". We can call either one a success, but we'll choose to be sexist towards guys and call a girl a success in this problem
2. The event is going to repeat 4 times, so we have 4 trials
3. The probability of a girl being born is 50% or 1/2 each time
4. We're interested in the probability that there are exactly 3 successes (3 girls)

The syntax here is `binompdf(trials, probability, value)`. In this case:

```
: binompdf(4, .5, 3)
```

This will give .25 when you run it, so there's a .25 (1/4) probability out of 4 children, 3 will be girls.

An alternate syntax for `binompdf()` leaves off the last argument, `value`. This tells the calculator to compute a list of the results for all values. For example:

```
: binompdf(4, .5)
```

This will come to `{.0625 .25 .375 .25 .0625}` when you run it. These are the probabilities of all 5 outcomes (0 through 4 girls) for 4 children with an equal probability of being born. There's a .0625 probability of no girls, a .25 probability of 1 girl, etc.

Advanced (for programmers)

The `binompdf()` and `binomcdf()` commands are the only ones apart from `seq()` that can return a list of a given length, and they do it much more quickly. It therefore makes sense, in some situations, to use these commands as substitutes for `seq()`.

Here's how to do it:

1. `cumSum(binomcdf(N,0))` gives the list `{1 2 ... N+1}`, and `cumSum(not(binompdf(N,0)))` gives the list `{0 1 2 ... N}`.
2. With `seq()`, you normally do math inside the list: for example, `seq(3I^2,I,0,5)`
3. With these commands, you do the same math outside the list: `3Ans^2` where `Ans` is the list `{0 1 ... 5}`.

An example:

```
: seq(2^I, I, 1, 5
can be
:cumSum(binomcdf(4, 0
:2^Ans
which in turn can be
:2^cumSum(binomcdf(4, 0
```

for a list of all values (0 to `trials`)
`binompdf(trials, probability)`

Menu Location

Press:

1. 2ND DISTR to access the distribution menu
2. 0 to select `binompdf()`, or use arrows.

Press ALPHA A instead of 0 on a TI-84+/SE with OS 2.30 or higher.

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

In general (where f() is some operation or even several operations):

```
:seq(f(I),I,1,N  
can be  
:cumSum(binomcdf(N-1,0  
:f(Ans  
which can sometimes be  
:f(cumSum(binomcdf(N-1,0
```

If the lower bound on I in the seq(statement is 0 and not 1, you can use binompdf(instead:

```
:seq(f(I),I,0,N  
can be  
:cumSum(not(binompdf(N,0  
:f(Ans  
which can sometimes be  
:f(cumSum(not(binompdf(N,0
```

This will not work if some command inside seq(can take only a number and not a list as an argument. For example, seq(L₁(I),I,1,5 cannot be optimized this way.

Formulas

The value of binompdf(is given by the formula

$$\text{binompdf}(n, p, k) = \binom{n}{k} p^k (1 - p)^{n-k} = \frac{n!}{k!(n-k)!} p^k (1 - p)^{n-k} \quad (1)$$

This formula is fairly intuitive. We want to know the probability that out of n trials, exactly k will be successes, so we take the probability of k successes - p^k - multiplied by the probability of (n-k) failures - $(1 - p)^{n-k}$ - multiplied by the number of ways to choose which k trials will be successes - $\binom{n}{k}$.

Error Conditions

- **ERR:DOMAIN** is thrown if the number of trials is at least 1 000 000 (unless the other arguments make the problem trivial)

Related Commands

- [binomcdf\(\)](#)
- [geometpdf\(\)](#)
- [geometcdf\(\)](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/binompdf>

The checkTmr(Command

The checkTmr(command is used together with the startTmr command to determine how much time has elapsed since the timer was started on the TI-84+/SE calculators. In particular, it returns the number of seconds since the built-in timer was started. An application of these commands is timing different commands or pieces of code, as well as countdowns in games, or a time-based score (such as in Minesweeper).

To use the timer, you first store startTmr to a variable (usually, a real variable) whenever you want the count to start. Now, whenever you want to check the elapsed time, you can use checkTmr(with the variable from above, giving you the number of seconds that have passed. Using checkTmr(doesn't stop the timer, you can do it as many times as you want to.

In the case of Minesweeper, for example, you would store startTmr to, for example, T, after setting up and displaying the board, display the result of checkTmr(T) in the game's key-reading loop, and store checkTmr(T) to the player's score if he wins.

Advanced Uses

To time a command or routine using startTmr and checkTmr(, use the following template:

```
:ClockOn  
:startTmr→T  
:For(A,1,(number)  
    (command(s) to be tested)  
:End  
:checkTmr(T)/(number)
```

Making (number) higher increases accuracy, but takes longer. Also, make sure not to modify the variables A or T inside the For(loop.

While this method eliminates human error from counting, it's prone to its own faults. A major one is that startTmr and checkTmr(always return whole numbers, but time is continuous. Depending on how close the start and end of the loop were to a clock tick, the number of seconds may be off by up to one second in either direction. To take this into account, you could replace the last line:

```
: (checkTmr(T)+{-1,1})/(number)
```

```
PROGRAM: TIMER  
:startTmr→A  
:For(B,1,randInt  
(≤2,5000  
:End  
:Disp checkTmr(A  
█
```

Command Summary

Returns the number of seconds since the timer was started.

Command Syntax

checkTmr(*Variable*)

Menu Location

This command can only be found in the catalog. Press:

1. 2nd CATALOG to enter the command catalog
2. C to skip to command starting with C
3. Scroll down to checkTmr(and select it

Calculator Compatibility

TI-84+/SE

Token Size

2 bytes

This will give you a list of the maximum and minimum possible times — the true time that the

command takes is guaranteed to be somewhere in between.

The other thing you need to be aware of when testing code is that there are many different things that will affect the time: the strength of the batteries, the amount of free RAM, and including the closing parenthesis on the For(loop. The last one, in particular, has an impact when using a lone If command or one of the IS>(and DS<(commands.

Related Commands

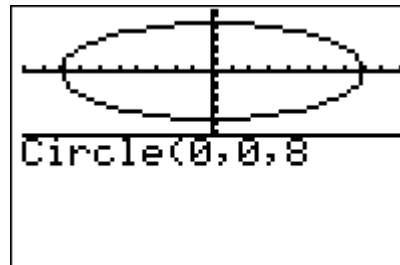
- startTmr

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/checktmr>

The Circle(Command

Circle(X, Y, r) will draw a circle at (X, Y) with radius r . X and Y will be affected by the window settings. The radius will also be affected by the window settings.

```
:Circle(5,5,5)
```



Advanced Uses

As you know, the radius is affected by the window settings. This means that if the x- and y-increment is two, the radius will be two pixels. However, there is another way to take advantage of this to draw ellipses. If the x- and y-increment are different, then the shape will not be a circle. For instance, with Xmin=0, Xmax=20, Ymin=0, and Ymax=31, Circle(10,10,2) will draw an ellipse, where the width is greater than the height. This concept is very interesting, but hard to explain, and I recommend that you experiment to try to understand it better.

Optimization

If a complex list such as $\{i\}$ is passed to Circle(as the fourth argument, the "fast circle" routine is used instead, which uses the symmetries of the circle to only do 1/8 of the trig calculations. For example:

```
:Circle(0,0,5  
can be  
:Circle(0,0,5,{i
```

Command Summary

Draws a circle.

Command Syntax

Circle(X, Y, r)

(83+ and higher only)
Circle($X, Y, r, \{i\}$)

Menu Location

Press:

1. Press [2ND] [PRGM] to enter the DRAW menu
2. Press [9] to insert Circle(

Calculator Compatibility

TI-83/84/+SE

Token Size

1 byte

Any list of complex numbers will work as the fourth argument in the same way, but there's no benefit to using any other list.

Command Timings

The ordinary Circle(is extremely slow. The fast circle trick discussed above cuts the time down to only about 30% of the "slow Circle(" time! While still not instant, this is faster than any replacement routine that can be written in TI-Basic.

For small radii, replace Circle(with Pt-On(instead.

Related Commands

- [Line\(](#)
- [Pt-On\(](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/circle>

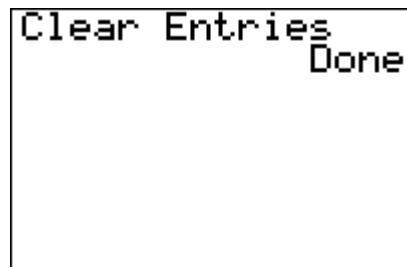
The Clear Entries Command

Normally, by pressing 2nd ENTER repeatedly, you can cycle through some of the recent entries on the home screen. With the Clear Entries command, this history is cleared (only Clear Entries remains in the history).

This can be used to free some memory, although it's recommended not to do this in a program (because clearing things without asking first isn't nice). Aside from that, maybe the only reason to use Clear Entries is to protect your privacy — although someone looking at your entries will know you cleared something, so it's not that effective.

Related Commands

- [ClrList](#)
- [ClrAllLists](#)
- [GarbageCollect](#)



Command Summary

Clears the history of commands previously entered on the homescreen.

Command Syntax

Clear Entries

Menu Location

Press:

1. 2nd MEM to access the memory menu.
2. 3 to select Clear Entries, or use arrows and ENTER.

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

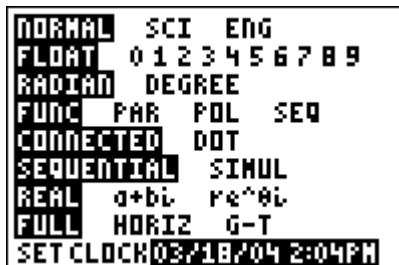
The ClockOff Command

The ClockOff command turns off the clock display at the bottom of the mode screen on the TI-84+/SE calculators. You can turn the clock back on by using the ClockOn command, or by selecting 'TURN CLOCK ON' ,displayed in place of the clock on the mode screen.

The ClockOff command does not actually turn the clock off. The time can still be accessed through use of the getTime and getDate commands, and all their cousins.

Related Commands

- ClockOn



```
NORMAL SCI ENG
FLOAT 0 1 2 3 4 5 6 7 8 9
RADIAN DEGREE
FUNC PAR POL SEQ
CONNECTED DOT
SEQUENTIAL SIMUL
REAL +bi Re^@l
FULL HORIZ G-T
SET CLOCK 03/18/04 2:04PM
```

Command Summary

Turns off the clock display in the mode screen.

Command Syntax

ClockOff

Menu Location

This command can only be found in the catalog. Press:

1. 2nd CATALOG to enter the command catalog
2. C to skip to command starting with C
3. Scroll down to ClockOff and select it

Calculator Compatibility

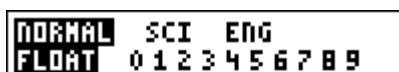
TI-84+/SE

Token Size

2 bytes

The ClockOn Command

The ClockOn command turns on the clock display at the bottom of the mode screen on the TI-84+/SE calculators. Alternatively, you can scroll down to the



```
NORMAL SCI ENG
FLOAT 0 1 2 3 4 5 6 7 8 9
RADIAN DEGREE
FUNC PAR POL SEQ
CONNECTED DOT
SEQUENTIAL SIMUL
REAL +bi Re^@l
FULL HORIZ G-T
SET CLOCK 03/18/04 2:04PM
```

'TURN CLOCK ON' message that is displayed in place of the clock on the mode screen and press ENTER twice. You can turn the clock off by using the [ClockOff](#) command.

Related Commands

- [ClockOff](#)

RADIAN	DEGREE		
FUNC	PAR	POL	SEQ
CONNECTED	DOT		
SEQUENTIAL	SIMUL		
REAL	a+bi	r^θi	
FULL	HORIZ	G-T	
SET CLOCK TURN CLOCK ON			

Command Summary

Turns on the clock display in the mode screen.

Command Syntax

ClockOn

Menu Location

This command can only be found in the catalog. Press:

1. 2nd CATALOG to enter the command catalog
2. C to skip to command starting with C
3. Scroll down to ClockOn and select it

Calculator Compatibility

TI-84+/SE

Token Size

2 bytes

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/clockon>

The ClrAllLists Command

The ClrAllLists command sets the dimension (length) of all lists to zero. This is virtually equivalent to deleting the lists, except for two differences:

- The lists still exist and will show up in the list menu and the memory management menu.
- The [dim\(](#) command will return 0 for a cleared list, rather than an error.

However, accessing a cleared list in any other way will return an error, just as for a deleted list.

The ClrAllLists command should **never** be used in a program you give to someone else or upload -

{1,2,3→L ₁	{1 2 3}
ClrAllLists	Done
dim(L ₁)	0

Command Summary

Sets the size of all defined lists to 0 (equivalent to applying the [ClrList](#) command to all defined lists)

unless the user is aware of this effect, they might lose important data stored in one of their lists. There is no way to limit the effect of ClrAllLists, so a program should use [ClrList](#) instead to avoid affecting unrelated lists (this is assuming you already want to use this questionably-useful effect).

Outside a program (or in a program for personal use), you might use this command to clear the contents of your lists to free up memory, while still not deleting the lists. This might possibly be convenient. Maybe.

Related Commands

- [ClrList](#)
- [DelVar](#)
- [dim\(](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/clralllists>

The ClrDraw Command

The ClrDraw command is useful clearing away something drawn on the graph screen; in particular, you want to do this at the beginning of a program that uses the graph screen, to get rid of anything that might be on it initially. If there are functions, plots, axes, labels, or grid enabled, these will be redrawn even after you ClrDraw. If you don't want these, you should turn them off before the ClrDraw command.

Like many other drawing commands, if you're outside a program and on the graph screen, you can use this command directly, without going to the home screen. Just select ClrDraw from the menu, and the screen will be cleared immediately.

Advanced Uses

Unless the final state of the graph screen is the intended effect of the program, you want to use ClrDraw at the end of the program so that the user doesn't have to deal with it.

Caution: if the graph screen is displayed even before you execute ClrDraw, the user variable Y will be reset to 0. This might be useful as a side effect, but

(applies to all defined lists).

Command Syntax

ClrDraw

Menu Location

Press:

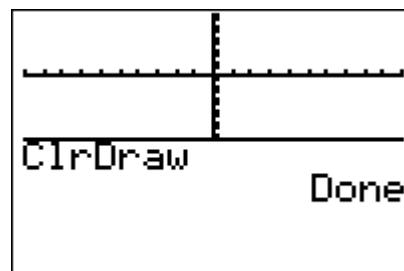
1. 2nd PRGM to enter the DRAW menu
2. 4 to select ClrDraw, or use arrows

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes



Command Summary

Clears the graph screen, redrawing functions, plots, and axes/grid/labels, if applicable.

Command Syntax

ClrDraw

Menu Location

Press:

1. 2nd PRGM to enter the DRAW menu
2. 1 or ENTER to select ClrDraw

it's more likely to turn out to be a nuisance if you were relying on Y to store something useful. Also, such a wacky effect might get removed in later OS versions, so it's a gamble relying on it to work for all users.

The RecallPic command does not erase what is previously on the graph screen when recalling a picture. Unless this is what you intend, use ClrDraw to erase the graph screen's old contents before recalling a picture.

Optimization

The ClrDraw command is not the only way to clear the screen. If something changes about the state of the functions or plots plotted on the graph, about the window dimensions, or the axes, grid, and labels, the graph screen will be marked as 'dirty' by the calculator, and will be cleared the next time you display it.

Don't be too confident about relying on this however. For example, if you cleared Y_1 before displaying the graph, and Y_1 previously contained something, the graph will be redrawn. However, if Y_1 never existed, then you haven't changed anything, and the graph will remain.

A lot of people choose their preferred window settings using the following two commands, which sets the window to $X = -47..47$, $Y = -31..31$:

```
ZStandard:ZInteger
```

Since this actually switches two window settings, at least one will be different from the previous settings, so the next time the graph screen is shown, it will be cleared without a ClrDraw command. There are other friendly window settings that you can use as well.

Related Commands

- ClrHome

See Also

- Friendly Graphing Window

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/clrdraw>

The ClrHome Command

There are numerous times in a program that you need a clear screen, so that you can display whatever text you want without it being interrupted. One place, in particular, is at the beginning of a program, since the previous program call(s) and any other text is typically still displayed on the screen. The simple ClrHome command is the command you

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

PROGRAM: EXAMPLE
:ClrHome
:Output(3,3,"EXAMPLE V1.1"
:Output(5,2,"BY TIBASICDEV")

use to clear the home screen.

When you use the `ClrHome`, it resets the cursor position to the top left corner of the home screen. This is what the `Disp` and `Pause` commands use as the reference for what line to display their text on, but it does not have any effect on `Output`.

Advanced Uses

You want to make sure to clear the home screen when exiting programs (at the end of a program). This ensures that the next program that the user runs will not have to deal with whatever text your program left behind. It also helps the user, because they will not have to manually clear the home screen by pressing the `CLEAR` key; you have already done it for them.

Error Conditions

- `ERR:INVALID` occurs if this statement is used outside a program.

Related Commands

- `ClrDraw`

See Also

- [Program Setup](#)
- [Program Cleanup](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/clrhome>

The `ClrList` Command

`ClrList` sets the length of a list (or several lists) to 0. This is virtually equivalent to deleting the list, except for two differences:

- The list still exists — it will be shown in the memory management menu and the list menu
- Calling the `dim` command on it will return 0, rather than an error.

In practice, there is rarely a reason to use this command over `DelVar`.

Advanced Uses

Command Summary

Clears the home screen of any text or numbers.

Command Syntax

`ClrHome`

Menu Location

While editing a program, press:

1. PRGM to enter the PRGM menu
2. RIGHT to enter the I/O menu
3. 8 to choose `ClrHome`, or use arrows

Calculator Compatibility

TI-83/84/+SE

Token Size

1 byte

```
{1,2,3} {1 2 3}
ClrList L1 Done
dim(L1) 0
```

Command Summary

Sets the dimension of a list or lists to 0.

Command Syntax

You might use ClrList when building up a list element by element and using dim(in the process:

```
:ClrList L1
:While 10>dim(L1
:Input X
:X→L1(1+dim(L1
:End
```

You may be thinking to yourself, "That is an extremely contrived example!" That's because that is an extremely contrived example. The truth is there is no good reason to use ClrList.

Optimization

Using DelVar instead of ClrList allows you to save a tiny bit of memory (between 12 and 16 bytes) that ClrList doesn't delete, while keeping almost every aspect of the list clearing the same.

Error Conditions

- **ERR:SYNTAX** is thrown if you leave off the L symbol when referring to a custom list (i.e., ClrList B will not work; you have to use ClrList LB).

Related Commands

- ClrAllLists
- DelVar
- dim(

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/clrlst>

The ClrTable Command

The ClrTable command clears all calculations for the table screen shown if you press 2nd TABLE. That is, all already-calculated values in the table are cleared, and TblInput is deleted. In IndpntAuto and DependAuto mode, this usually isn't noticeable because the table will be recalculated almost immediately when you next look at it (unless one of the entered functions is so complicated it takes a while to calculate it). This mainly has an effect in IndpntAsk or DependAsk mode, in which case the corresponding parts of the table will be cleared entirely.

Advanced Uses

Command Syntax

ClrTable /list1, //list3//, ...

Menu Location

Press:

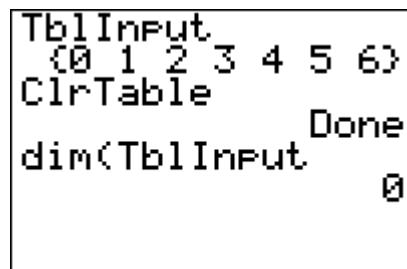
1. STAT to access the statistics menu
2. 4 to select ClrList, or use arrows.

Calculator Compatibility

TI-83/84/+SE

Token Size

1 byte



```
TblInput
{0 1 2 3 4 5 6}
ClrTable      Done
dim(TblInput) 0
```

Command Summary

Clears saved calculations for the table screen.

Command Syntax

As a side effect, ClrTable seems to have all the effects of ClrDraw — it clears the graph screen, and any equations or plots will be regraphed when next the graph screen is displayed.

Command Timings

ClrTable and ClrDraw take the same amount of time to clear the screen.

Related Commands

- ClrDraw
- DispGraph
- DispTable

ClrTable

Menu Location

While editing a program, press:

1. PRGM to access the program menu.
2. RIGHT to access the I/O submenu.
3. 9 to select ClrTable.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/clrtable>

The \wedge Command

The \wedge operator is used to raise a number to a power. It can be used with numbers, expressions, and lists. It can be used for taking nonnegative integer powers of square matrices (up to the 255th power only, however), but not for negative powers (use $\frac{-1}{}$ instead) or matrix exponentials (which the TI-83+ cannot do).

In general, x^y returns the same results as $e^{(y \ln(x))}$. For expressions of the form $x^{(p/q)}$, where p and q are integers and q is an odd number, the principal branch is returned if x is complex, but the real branch is returned if x is a negative real number.

```
(-1)^(1/3)
      -1
(-1+0i)^(1/3)
      .5+.8660254038i
```

```
5^6          15625
(52X)^8        0
{1,5,9}^3      {1 125 729}
```

Command Summary

Raises a number to a power.

Command Syntax

x^y

Menu Location

Press [\wedge]

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

Optimization

When raising 10 or e to a power, use the 10 \wedge (and $e\wedge$ (commands instead. Similarly, use the $\frac{2}{}, \frac{3}{},$ or $\frac{-1}{}$ commands for raising a number to the 2, 3, or -1 power.

Error Conditions

- **ERR:DOMAIN** is thrown when calculating 0^0 , or raising 0 to a negative power.
- **ERR:NONREAL ANS** is thrown in Real mode if the result is complex (and the operands are real)

Related Commands

- *, /, \sqrt{x}
- 10^x , e^x

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/power>

The $\sqrt[x]{ }$ Command

This command takes the x th root of a number. If used on a list, it will return a list with the x th root of each element. Also valid are the forms $list^{\sqrt[x]{}}$ and $list1^{\sqrt[x]{}}list2$.

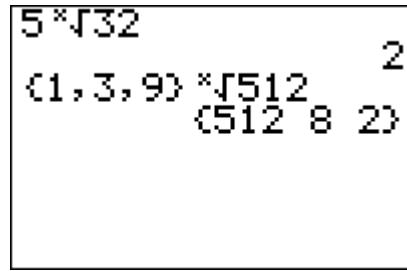
```
:2 $\sqrt[4]{}$ 
      2
:5 $\sqrt[2]{}$ 
      1.148698355
:3 $\sqrt[3]{\{1, -8, 27\}}$ 
      {1   -2   3}
:{3,2} $\sqrt[3]{\{8,9\}}$ 
      {2   3}
Real mode:
:4 $\sqrt[-1]{}$ 
      <returns error>
a+b i mode:
:4 $\sqrt[-1]{}$ 
      .7071067812+.7071067812i
```

See the notes on the $\sqrt[x]{}$ command for an explanation on how $\sqrt[x]{}$ behaves depending on whether its input is real or complex.

Optimization

If you want to take the second or third root of a number, use $\sqrt[2]{}$ or $\sqrt[3]{}$ instead.

```
:2 $\sqrt[X]{}$ 
```



5 $\sqrt[3]{32}$
2
(1,3,9) $\sqrt[5]{512}$ 2
(512 8 2)

Command Summary

Takes the x th root of an input.

Command Syntax

$A^{\sqrt[x]{B}}$

Menu Location

While editing a program, press:

1. MATH to open the math menu
2. 5 or use the arrow keys to select

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

can be
: \sqrt{X}

Error Conditions

- **ERR:NONREAL ANS** if you try to take an even root of a negative number or list element in Real mode.

Related Commands

- $\sqrt{ }$
- $\sqrt[3]{ }$
- \wedge

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/xroot>

The \neq Command

The \neq (not equal) operator takes two numbers, variables, or expressions, and tests to see if they are not equal to each other. It will return 1 if they are not, and 0 if they are. When determining the order of operations, \neq will be executed after the math operators, but it will be executed before the logical operators and in the order that it appears from left to right with the other relational operators.

```
:1≠0  
1  
:DelVar X3→Y  
:X≠Y  
1
```

Advanced Uses

Just like the other relational operators, \neq can take real numbers and lists for variables. In order to compare the lists, however, both must have the same dimensions; if they don't, the calculator will throw a ERR:DIM MISMATCH error. When comparing a real number to a list, the calculator will actually compare the number against each element in the list and return a list of 1s and 0s accordingly.

```
:{2,4,6,8}≠{1,3,5,7  
{1 1 1 1}  
:5≠{1,2,3,4,5
```

2+2≠4	0
2+2≠5	1
{1,2}≠1	{0 1}

Command Summary

Returns true if value1 is not equal to value2.

Command Syntax

value1 \neq *value2*

Menu Location

Press:

1. 2nd TEST to access the test menu.
2. 2 to select \neq , or use arrows.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

```
{1 1 1 1 0}
```

Besides real numbers and lists, \neq also allows you compare strings, matrices, and complex numbers. However, the variables must be of the same type, otherwise the calculator will throw a ERR:DATA TYPE error; and just like with lists, both matrices must have the same dimensions, otherwise you will get a ERR:DIM MISMATCH error.

```
: [[1,2,3]]≠[[1,2,3  
          0  
: "HELLO"≠"WORLD  
          1  
: (3+4i)≠(5-2i)      (the parentheses are added for clarity)  
          1
```

Optimization

Because the calculator treats every nonzero value as true and zero as false, you don't need to compare if a variable's value is nonzero. Instead, you can just put the variable by itself.

```
: If C≠0  
can be  
: If C
```

If you are struggling to understand why that makes sense, an easy way to look at it is that it is just another form of subtraction. The statement will only be false if C is equal to zero, since $0-0=0$. But if C is something else, such as 5, then $5-0=5$, which is true. This isn't really an optimization, except for when the number you are subtracting is zero.

```
: If A≠5  
can be  
: If A-5
```

Error Conditions

- ERR:DATA TYPE is thrown if you try to compare two different kinds of variables, such as a string and number or a list and matrix.
- ERR:DIM MISMATCH is thrown if you try to compare two lists or matrices that have different dimensions.

Related Commands

- \equiv (equal)
- \geq (greater than)
- \geq (greater than equal)
- \leq (less than)
- \leq (less than equal)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/notequal>

The < Command

The < (less than) operator takes two numbers, variables, or expressions, and tests to see if the first one has a value less than the second one. It will return 1 if it is less than, and 0 if it is not. When determining the order of operations, < will be executed after the math operators, but it will be executed before the logical operators and in the order that it appears from left to right with the other relational operators.

```
:1<0  
0  
  
:DelVar X3→Y  
:X<Y  
1
```

Advanced Uses

Just like the other relational operators, < can take real numbers and lists for variables. In order to compare the lists, however, both must have the same dimensions; if they don't, the calculator will throw a ERR:DIM MISMATCH error. When comparing a real number to a list, the calculator will actually compare the number against each element in the list and return a list of 1s and 0s accordingly.

```
:{2,4,6,8}<{1,3,5,7  
{0 0 0 0}  
:5<{1,2,3,4,5  
{0 0 0 0 0}
```

Unfortunately, < does not work with strings, matrices, or complex numbers (only \equiv and \neq do), and the calculator will actually throw a ERR:DATA TYPE error if you try to compare them. In the case of strings, however, it should be pretty obvious why: a string represents a sequence of characters, and does not associate a value to any character, so there is nothing to compare.

Error Conditions

- ERR:DATA TYPE is thrown if you try to compare strings, matrices, or complex numbers.
- ERR:DIM MISMATCH is thrown if you try to compare two lists that have different dimensions.

Related Commands

- \equiv (equal)

0<1	1
1<0	0
{1,4}<{2,3}	{1 0}

Command Summary

Returns true if value1 is less than value2.

Command Syntax

value1<*value2*

Menu Location

Press:

1. 2nd TEST to access the test menu.
2. 5 to select <, or use arrows.

Calculator Compatibility

TI-83/84/+SE

Token Size

1 byte

- \neq (not equal)
- \geq (greater than)
- \geq (greater than equal)
- \leq (less than equal)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/lessthan>

The \leq Command

The \leq (less than equal) operator takes two numbers, variables, or expressions, and tests to see if the first one has a value less than or equal to the second one. It will return 1 if it is less than or equal to, and 0 if it is not. When determining the order of operations, \leq will be executed after the math operators, but it will be executed before the logical operators and in the order that it appears from left to right with the other relational operators.

```
:1≤0
0
:DelVar X3→Y
:X≤Y
1
```

Advanced Uses

Just like the other relational operators, \leq can take real numbers and lists for variables. In order to compare the lists, however, both must have the same dimensions; if they don't, the calculator will throw a ERR:DIM MISMATCH error. When comparing a real number to a list, the calculator will actually compare the number against each element in the list and return a list of 1s and 0s accordingly.

```
:{2,4,6,8}≤{1,3,5,7
{0 0 0 0}
:5≤{1,2,3,4,5
{0 0 0 0 1}
```

Unfortunately, \leq does not work with strings, matrices, or complex numbers (only \equiv and \neq do), and the calculator will actually throw a ERR:DATA TYPE error if you try to compare them. In the case of strings, however, it should be pretty obvious why: a string represents a sequence of characters, and does not associate a value to any character, so there is nothing to compare.

Error Conditions

$1 \leq 0$	0
$1 \leq 2$	1
$\{-1, 0, 1, 2\} \leq 0$	$(1 \ 1 \ 0 \ 0)$

Command Summary

Returns true if value1 is less than or equal to value2.

Command Syntax

value1 \leq *value2*

Menu Location

Press:

1. 2nd TEST to access the test menu.
2. 6 to select \leq , or use arrows.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

- **ERR:DATA TYPE** is thrown if you try to compare strings, matrices, or complex numbers.
- **ERR:DIM MISMATCH** is thrown if you try to compare two lists that have different dimensions.

Related Commands

- \equiv (equal)
- \neq (not equal)
- \geq (greater than)
- $\geq\geq$ (greater than equal)
- \leq (less than)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/lessthanorequal>

The = Command

The = (equal) operator takes two numbers, variables, or expressions, and tests to see if they are equal to each other. It will return 1 if they are, and 0 if they are not. When determining the order of operations, = will be executed after the math operators, but it will be executed before the logical operators and in the order that it appears from left to right with the other relational operators.

```
: 1=0
      0
: DelVar X3→Y
: X=Y
      0
```

Advanced Uses

Just like the other relational operators, = can take real numbers and lists for variables. In order to compare the lists, however, both must have the same dimensions; if they don't, the calculator will throw a ERR:DIM MISMATCH error. When comparing a real number to a list, the calculator will actually compare the number against each element in the list and return a list of 1s and 0s accordingly.

```
: {2,4,6,8}={1,3,5,7
      {0 0 0 0}
: 5={1,2,3,4,5
      {0 0 0 0 1}
```

2+2=4	1
2+2=5	0
{1,2}={3,2}	{0 1}

Command Summary

Returns true if value1 is equal to value2.

Command Syntax

value1=*value2*

Menu Location

Press:

1. 2nd TEST to access the test menu.
2. 1 to select =, or use arrows.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

Besides real numbers and lists, = also allows you compare strings, matrices, and complex

numbers. However, the variables must be of the same type, otherwise the calculator will throw a [ERR:DATA TYPE](#) error; and just like with lists, both matrices must have the same dimensions, otherwise you will get a [ERR:DIM MISMATCH](#) error.

```
: [[1,2,3]]=[[1,2,3
              1
: "HELLO"="WORLD
              0
: (3+4i)=(5-2i)      (the parentheses are added for clarity)
              0
```

Optimization

When the only values that are possible for a variable are 1 and 0, you can get rid of the = sign and simply use the variable by itself.

```
: If X=1
can be
: If X
```

Error Conditions

- [ERR:DATA TYPE](#) is thrown if you try to compare two different kinds of variables, such as a string and number or a list and matrix.
- [ERR:DIM MISMATCH](#) is thrown if you try to compare two lists or matrices that have different dimensions.

Related Commands

- \neq (not equal)
- $>$ (greater than)
- \geq (greater than equal)
- $<$ (less than)
- \leq (less than equal)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/equal>

The \geq Command

The \geq (greater than equal) operator takes two numbers, variables, or expressions, and tests to see if the first one has a value greater than or equal to the second one. It will return 1 if it is greater than or equal to, and 0 if it is not. When determining the order of operations, \geq will be executed after the [math](#) operators, but it will be executed before the [logical](#) operators and in the order that it appears from left to right with the other [relational](#) operators.

```
1≥0          1
1≥2          0
{-1,0,1,2}≥0 {0 1 1 1}
```

```
:1≥0  
1  
  
:DelVar X3→Y  
:X≥Y  
0
```

Advanced Uses

Just like the other relational operators, \geq can take real numbers and lists for variables. In order to compare the lists, however, both must have the same dimensions; if they don't, the calculator will throw a ERR:DIM MISMATCH error. When comparing a real number to a list, the calculator will actually compare the number against each element in the list and return a list of 1s and 0s accordingly.

```
:{2,4,6,8}≥{1,3,5,7  
{1 1 1 1}  
:5≥{1,2,3,4,5  
{1 1 1 1 1}
```

Unfortunately, \geq does not work with strings, matrices, or complex numbers (only \equiv and $\not\equiv$ do), and the calculator will actually throw a ERR:DATA TYPE error if you try to compare them. In the case of strings, however, it should be pretty obvious why: a string represents a sequence of characters, and does not associate a value to any character, so there is nothing to compare.

Error Conditions

- ERR:DATA TYPE is thrown if you try to compare strings, matrices, or complex numbers.
- ERR:DIM MISMATCH is thrown if you try to compare two lists that have different dimensions.

Related Commands

- \equiv (equal)
- $\not\equiv$ (not equal)
- \geq (greater than)
- \leq (less than)
- \leqq (less than equal)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/greaterthanequal>

The $>$ Command

The $>$ (greater than) operator takes two numbers, variables, or expressions, and tests to see if the first

Command Summary

Returns true if value1 is greater than or equal to value2.

Command Syntax

value1 \geq *value2*

Menu Location

Press:

1. 2nd TEST to access the test menu.
2. 4 to select \geq , or use arrows.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

$1 > 0$

one has a value greater than the second one. It will return 1 if it is greater, and 0 if it is not. When determining the order of operations, $>$ will be executed after the math operators, but it will be executed before the logical operators and in the order that it appears from left to right with the other relational operators.

```
:1>0  
1  
  
:DelVar X3→Y  
:X>Y  
0
```

Advanced Uses

Just like the other relational operators, $>$ can take real numbers and lists for variables. In order to compare the lists, however, both must have the same dimensions; if they don't, the calculator will throw a ERR:DIM MISMATCH error. When comparing a real number to a list, the calculator will actually compare the number against each element in the list and return a list of 1s and 0s accordingly.

```
:{2,4,6,8}>{1,3,5,7  
{1 1 1 1}  
:5>{1,2,3,4,5  
{1 1 1 1 0}
```

Unfortunately, $>$ does not work with strings, matrices, or complex numbers (only \equiv and \neq do), and the calculator will actually throw a ERR:DATA TYPE error if you try to compare them. In the case of strings, however, it should be pretty obvious why: a string represents a sequence of characters, and does not associate a value to any character, so there is nothing to compare.

Error Conditions

- ERR:DATA TYPE is thrown if you try to compare strings, matrices, or complex numbers.
- ERR:DIM MISMATCH is thrown if you try to compare two lists that have different dimensions.

Related Commands

- \equiv (equal)
- \neq (not equal)
- \geq (greater than equal)
- \leq (less than)
- \leq (less than equal)

0>1	1
{2, -2, 3}>0	0
	{1 0 1}

Command Summary

Returns true if value1 is greater than value2.

Command Syntax

value1 $>$ *value2*

Menu Location

Press:

1. 2nd TEST to access the test menu.
2. 3 to select $>$, or use arrows.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

The - Command

The - (subtract) operator takes two numbers, variables, or expressions and subtracts one from the other, thus returning the difference between them. The - operator appears lower in the order of operations than both * and /, so if those appear in an expression, they will be executed first. In addition, the ± operator has the same order of operations as -, so the calculator simply executes them left to right in the order that they appear.

```
: 1 - 1  
0  
  
: 5 → X  
: 2 - 3X  
- 13  
  
: 2 → A : 3 → B  
: A / B - B / A  
- . 8333333333
```

```
E3 - (-5, -92  
(995 1092)  
[[2, 1][3, 4]] - [[6  
, 5][ -3, 8]]  
[[ -4 -4]  
[6 -4]]  
5/3 - 3/5  
1.066666667
```

Command Summary

Returns the difference between two numbers.

Command Syntax

value1 - value2

Menu Location

Press [-]

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

Error Conditions

- **ERR:SYNTAX** is thrown if you try to use - (subtract) in place of _ (negative). Because they look very similar, it's easy to get this error; at the same time, it's an easy error to fix.

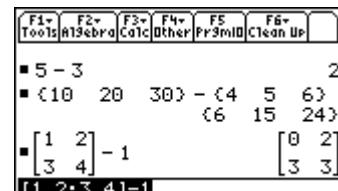
Related Commands

- + ([add](#))
- * ([multiply](#))
- / ([divide](#))

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/subtract>

The - Command

The - (subtract) operator takes two numbers, variables, or expressions and subtracts one from the other, thus returning the difference between them. The - operator appears lower in the order of operations than both * and /, so if those appear in an expression, they will be executed first. In addition, the + operator has the same order of operations as -



, so the calculator simply executes them left to right in the order that they appear.

MAIN RAD AUTO FUNC 3/20

```
: 1 - 1  
0  
  
: 5 → X  
: 2 - 3 X  
- 13  
  
: 2 → A : 3 → B  
: A / B - B / A  
- . 8333333333
```

Related Commands

- \pm (add)
- $*$ (multiply)
- $/$ (divide)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/68k:subtract>

The ! Command

$!$ is the factorial function, where $n! = n*(n-1)!$ and $0! = 1$, n an nonnegative integer. The function also works for arguments that are half an odd integer and greater than $-1/2$: $(-\frac{1}{2})!$ is defined as $\sqrt{\pi}$ and the rest are defined recursively.

```
3!  
6  
(-.5)!  
1.772453851  
Ans2  
3.141592654
```

The combinatorial interpretation of factorials is the number of ways to arrange n objects in order.

Error Conditions

- **ERR:DOMAIN** for any numbers except the ones mentioned above.

Command Summary

Returns the difference of two numbers.

Command Syntax

value1 - *value2*

Menu Location

Press [-] to paste -

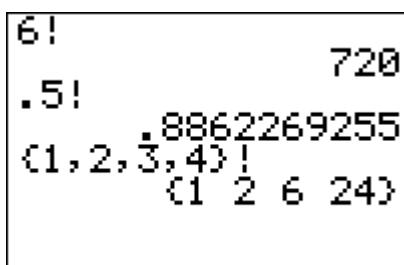
Calculator Compatibility

This command works on all calculators.

Token Size

1 byte total:

- 0x8D (command identifier)



6!
720
.5!
.8862269255
{1,2,3,4}!
(1 2 6 24)

Command Summary

Calculates the factorial of a number or list.

Command Syntax

value!

Menu Location

Press:

1. MATH to access the math menu.

Related Commands

- [nPr](#)
- [nCr](#)

2. LEFT to access the PRB submenu.
3. 4 to select !, or use arrows.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/factorial>

The / Command

The / (divide) operator takes two numbers, variables, or expressions and divides them, thus returning a single new value. The / operator appears higher in the order of operations than both \pm and $_$, so if those appear in an expression, / will be executed first. In addition, the $*$ operator has the same order of operations as /, so the calculator simply executes them left to right in the order that they appear.

```
: 1 / 1  
1  
  
: 5 → X  
: 2 / 3 X  
.0571428571  
  
: 2 → A : 3 → B  
: A / B / B / A  
.1111111111
```

X^Y / (Z + 0)
.9090909091
1 / (2, 3, 4)
(.5 .333333333...
5 / 3 / 5 / 3
.1111111111

Command Summary

Returns the division of two numbers.

Command Syntax

value1 / value2

Menu Location

Press [/]

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

Related Commands

- [+ \(add\)](#)
- [- \(subtract\)](#)
- [* \(multiply\)](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/divide>

The * Command

The `*` (multiply) operator takes two numbers, variables, or expressions and multiplies their values together, thus returning a single new value. The `*` operator appears higher in the order of operations than both `±` and `÷`, so if those appear in an expression, `*` will be executed first. In addition, the `/` operator has the same order of operations as `*`, so the calculator simply executes them left to right in the order that they appear.

```
: 1 * 1  
1  
  
: 5 → X  
: 2 * 3 X  
30  
  
: 2 → A : 3 → B  
: A / B * B / A  
1
```

Advanced Uses

As it turns out, the most advanced way to use `*` is by not using it at all. The TI-83 series of calculators does implicit multiplication, meaning it automatically recognizes when you want to multiply, so there is often no need to use `*`.

```
: 5 * A → B  
can be  
: 5 A → B
```

There are a few cases in which omitting the multiplication sign doesn't work. For example, $2^4 \cdot 3$ (which evaluates to 16000) can't be replaced by $2^4_E 3$, since the latter is interpreted as $2^4(4000)$.

Optimization

The `*` sign has the same truth value as the `and` operator because they both return zero if one or more of the numbers is zero (based on Boolean logic). Consequently, you sometimes see people implicitly multiplying expressions together in conditionals and loops, instead of joining them together with `and`. Unfortunately, this is not only usually larger in size, but often times slower.

```
: If (A=2)(B=7)  
should be  
: If A=2 and B=7
```

It does save some space when you can avoid using parentheses:

`X Y(Z+θ)` 110
`3<2,3,4` {6 9 12}
`{3,6,9} [A]{1,2}` {9 18 27}

Command Summary

Returns the multiplication of two numbers.

Command Syntax

`value1 * value2`

Menu Location

Press `[*`

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

```
:If A and B  
could be  
:If AB
```

Related Commands

- + ([add](#))
- - ([subtract](#))
- / ([divide](#))

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/multiply>

The % Command

The % symbol is an undocumented command on the TI-83 series calculators starting with OS version 1.15. It's useful as a shortcut for percents - it divides by 100, so it will convert numbers to percentages. For example, 50% will become 50/100 or 1/2, which is just what 50% should be.

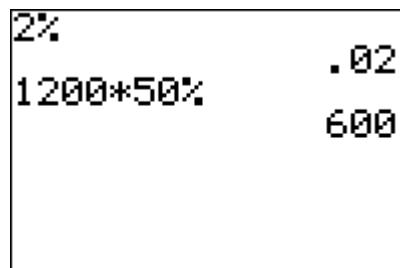
Although this trick can save you a few bytes, it also makes your program incompatible with old OS versions — it's up to you to decide if the tradeoff is worth it.

Error Conditions

- [ERR:INVALID](#) is thrown on older operating system versions.

Related Commands

- [sub\(](#)



2%
1200*50% .02
600

Command Summary

Short for dividing by 100.

Command Syntax

value%

Menu Location

This command can only be accessed through a hex editor (its hex code is 0xBB 0xDA)

Calculator Compatibility

TI-83/84+/SE, OS v1.15+

Token Size

2 bytes

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/percent>

The + Command

The + (add) operator takes two numbers, variables, or expressions and adds their values together, thus returning a single new value. The + operator appears lower in the order of operations than both `*` and `_`, so if those appear in an expression, they will be executed first. In addition, the `_` operator has the same order of operations as +, so the calculator simply executes them left to right in the order that they appear.

```
: 1+1  
2  
  
: 5→X  
: 2+3X  
17  
  
: 2→A : 3→B  
: A/B+B/A  
2.166666667
```

Advanced Uses

The + operator is overloaded (meaning it has more than one function) by the calculator, and it can be used to put strings together. The strings can consist of whatever combination of text and characters that you want, but it unfortunately does not allow you to join a string to a number (i.e., "Hello5" cannot be made with "Hello"+5).

```
: "HELLO"+"WORLD  
"HELLOWORLD  
  
: "TI"+" - "+"BASIC  
"TI-BASIC
```

Related Commands

- - ([subtract](#))
- * ([multiply](#))
- / ([divide](#))

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/add>

6+(2sin(X),3cos(Y),8tan(Z)
(4.082151451 4...
(3X)+(2Y)

19

"SITE: "+"U+i
SITE: U+i

Command Summary

Returns the sum of two numbers, or joins two strings together.

Command Syntax

value1 + *value2*

string1 + *string2*

Menu Location

Press [+]

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

The + Command

The + (add) operator takes two numbers, variables, or expressions and adds their values together, thus returning a single new value. The + operator appears lower in the order of operations than both * and /, so if those appear in an expression, they will be executed first. In addition, the - operator has the same order of operations as +, so the calculator simply executes them left to right in the order that they appear.

```
:1+1  
2  
  
:5→X  
:2+3*X  
17  
  
:2→A :3→B  
:A/B+B/A  
2.16666667
```

Related Commands

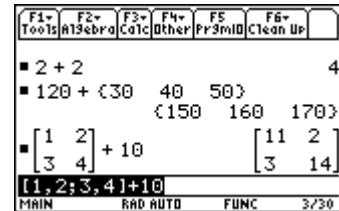
- - (subtract)
- * (multiply)
- / (divide)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/68k:add>

The → Command

The → (store) command will store a number, variable, or expression to a variable, using the respective value(s) of the variable(s) at the time. When storing a value in a variable, you have the value on the left side of → and the variable that it will be stored to on the right side.

```
:1→X  
1  
  
{1.3,5.7,9.11→ABC  
{1.3 5.7 9.11}
```



Command Summary

Returns the sum of two numbers.

Command Syntax

value1 + value2

Menu Location

Press [+]

Calculator Compatibility

This command works on all calculators.

Token Size

1 byte total:

- 0x8B (command identifier)

```
23→X  
23  
seq(I,I,1,5→L1  
{1 2 3 4 5}  
"TIBASICDEV→Str1
```

TIBASICDEV

Command Summary

Stores a value to a variable.

Command Syntax

```
: "HELLO WORLD→Str1  
    "HELLO WORLD"
```

Advanced

It's not easy to put a → symbol into a string, since "→→Str1 would produce a syntax error (and in general, when the calculator 'sees' a → symbol, it assumes that the string is over, and interprets the symbol literally).

However, you can use Equ►String((outside a program) to get the → or " symbols in a string:

1. Type them on the home screen and press [ENTER]
2. Select 1:Quit when the ERR:SYNTAX comes up.
3. Press [Y=] to go to the equation editor.
4. Press [2nd] [ENTRY] to recall the symbols to Y_1
5. Now, use $\text{Equ}\blacktriangleright\text{String}(Y_1, \text{Str1})$ to store the symbols to a string.

Optimization

You can remove closing parentheses, braces, brackets, and quotes that are before a → command.

```
: "Hello"→Str1  
can be  
: "Hello→Str1
```

Related Commands

- DelVar

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/store>

The \neg Command

The \neg (negative) operator takes one number, variable, or expression and negates its value, thus returning the negative equivalent of it. The \neg operator appears higher in the order of operations than both the relational and logical operators, so it will be executed first. In addition, it has the same order of operation as the other math operators, so the calculator simply executes them left to right in the order that they appear.

```
:  $\neg$  1
```

Value→Variable

Menu Location

Press [STO►]

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

```
-5X(-9(-3Y  
-6210  
-seq(I2, I, 1, 4  
(-1 -4 -9 -16)  
-X-3  
-20
```

Command Summary

Returns the negative value of a

```

: 5→X
: -3(X+2
      -21
: -2→A: -3→B
: AB
      6

```

Returns the negative value of a number.

Command Syntax

`-value`

Menu Location

Press [(−)]

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

```

: -A+B→C
can be
: B-A→C

```

Error Conditions

- **ERR:SYNTAX** is thrown if you try to use `¬` (negative) in place of `-` (subtract). Because they look very similar, it's easy to get this error; at the same time, it's an easy error to fix.

Related Commands

- `+` ([add](#))
- `-` ([subtract](#))
- `*` ([multiply](#))
- `/` ([divide](#))

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/negative>

The `L` Command

The `L` command indicates the beginning of a custom list (i.e., any list the user creates, not including the default lists `L1...L6`). You almost always need to include this when accessing or manipulating a custom list.

Optimization

You don't need to include the `L` command when [storing](#) (→) to a list. Some of the list commands also allow for this optimization, such as [SetUpEditor](#).

```

{1,5,.2,8→L1
 {1 5 .2 8}
 {5,2,1,.8→LX
 {5 2 1 .8}
 LX(1→L1(1
      5

```

Command Summary

Indicates the beginning of a custom

However, it can create problems when using Input and Prompt because you might only be asking the user to input a list, but a real variable would also be allowed.

Error Conditions

- **ERR:UNDEFINED** is thrown if you try to use L on an undefined list.

Related Commands

- \rightarrow (store)

Indicates the beginning of a custom list.

Command Syntax

L LISTNAME

Menu Location

While editing a program, press:

1. 2nd LIST to access the List menu
2. RIGHT to access the OPS submenu
3. 2nd B to select L , or use arrows

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/l>

The $\sqrt{}$ Command

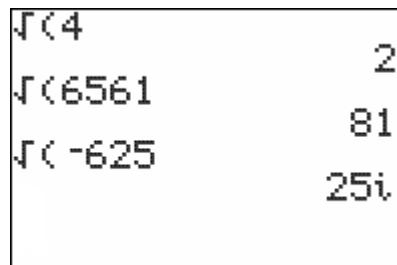
Takes the square root of a positive or negative number. It works exactly the same as $2^{\times}\sqrt{}$ or $^{\wedge}(1/2)$ but is smaller and uses an ending parenthesis. If used on a list, it will return a list with the square root of each element.

```
 $\sqrt{4}$ 
2
 $\sqrt{2}$ 
1.414213562
 $\sqrt{\{1, -1\}}$ 
{1 i}
```

This may return a complex number or throw **ERR:NONREAL ANS** (depending on mode settings) if taking the square root of a negative number.

Optimization

Never raise something to the one-half power



A screenshot of a TI-84 calculator displaying four examples of the $\sqrt{}$ command. The first example, $\sqrt{4}$, returns the value 2. The second example, $\sqrt{6561}$, returns the value 81. The third example, $\sqrt{-625}$, returns the value $25i$. The fourth example is partially visible at the bottom of the screen.

Command Summary

Take the square root of a number.

Command Syntax

$\sqrt{(input)}$

Menu Location

Press 2nd $\sqrt{}$ to paste the $\sqrt{}$ command.

Calculator Compatibility

explicitly; use this command instead.

```
:X^(1/2)→X  
can be  
:√(X→X)
```

TI-83/84/+SE

Token Size

1 byte

Error Conditions

- **ERR:NONREAL ANS** when taking the square root of a negative number in Real mode.

Related Commands

- \wedge
- \sqrt{x}
- $\sqrt[3]{x}$
- R►Pr(

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/square-root>

The conj(Command

conj(z) returns the complex conjugate of the complex number z. If z is represented as $x+iy$ where x and y are both real, conj(z) returns $x-iy$. Also works on a list of complex numbers.

```
conj(3+4i)  
3-4i
```

```
conj(3  
3  
conj(3+2i  
3-2i  
(3+2i)conj(3+2i  
13
```

The conjugate of a number z is often written \bar{z} , and is useful because it has the property that $z\bar{z}$ and $z + \bar{z}$ are real numbers.

Related Commands

- abs(
- angle(
- real(
- imag(

Command Summary

Calculates the complex conjugate of a complex number.

Command Syntax

conj(*value*)

Menu Location

Press:

1. MATH to access the math menu.
2. RIGHT, RIGHT to access the CPX (complex) submenu
3. ENTER to select conj(.

Calculator Compatibility

TI-83/84/+SE

Token Size

2 bytes

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/conj>

The Connected Command

The Connected command sets all equations to use the usual graph style - a connected line. In addition, this graph style is made the default, so that when a variable is deleted it will revert to this graph style. The other possible setting for this option is Dot.

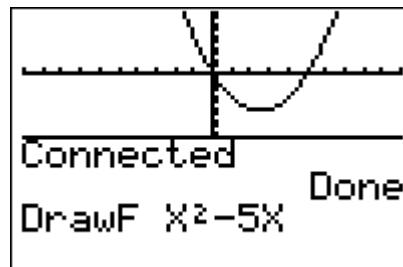
Compare this to the GraphStyle(command, which puts a single equation into a specified graph style.

The Connected and Dot commands don't depend on graphing mode, and will always affect all functions, even in other graphing modes. The exception to this is that sequence mode's default is always the dotted-line style, even when Connected mode is set. The Connected command will still change their graphing style, it just won't change the default they revert to.

In addition to graphing equations, this setting also affects the output of DrawF, DrawInv, and Tangent(.

Related Commands

- Dot
- GraphStyle(



Command Summary

Sets all equations to use the connected graphing style, and makes it the default setting.

Command Syntax

Connected

Menu Location

Press:

1. MODE to access the mode menu.
2. Use arrows to select Connected.

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/connected>

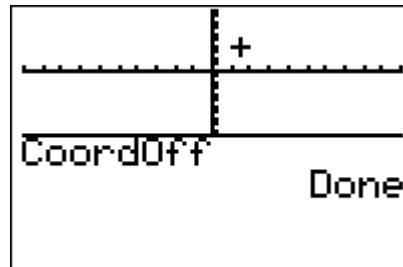
The CoordOff Command

When moving a cursor on a screen, it's possible for the calculator to display the coordinates of the current point (either polar or rectangular coordinates, depending on which of RectGC or PolarGC is set). The CoordOff command turns off this option.

To turn off this option, use the CoordOn command.

Related Commands

- CoordOn
- RectGC
- PolarGC



Command Summary

Turns off the cursor coordinate display on the graph screen.

Command Syntax

`CoordOff`

Menu Location

Press:

1. 2nd FORMAT to access the graph format menu
2. Use arrows and ENTER to select CoordOff

Calculator Compatibility

TI-83/84/+/SE

Token Size

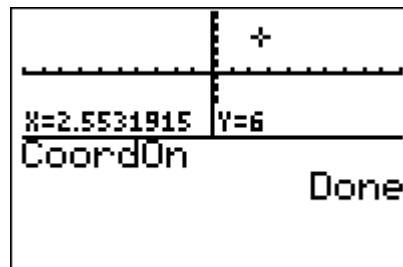
2 bytes

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/coordoff>

The CoordOn Command

When moving a cursor on a screen, it's possible for the calculator to display the coordinates of the current point (either polar or rectangular coordinates, depending on which of RectGC or PolarGC is set). The CoordOn command turns on this option (to disable it, use the CoordOff command).

The coordinates are displayed in practically every situation when you're moving a cursor on the graph screen, including even the Trace, Input or Select commands in a program. The interactive mode of Text(and the Pen tool are the exceptions — this is because these two situations involve a pixel



Command Summary

Turns on the cursor coordinate display on the graph screen.

coordinate, and not a point.

Related Commands

- [CoordOff](#)
- [RectGC](#)
- [PolarGC](#)

Command Syntax

CoordOn

Menu Location

Press:

1. 2nd FORMAT to access the graph format menu
2. Use arrows and ENTER to select CoordOn

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/coordon>

The \cos^{-1} (Command

\cos^{-1} (returns the arccosine of its argument. It is the inverse of [cos](#)(, which means that $\cos^{-1}(n)$ produces an angle θ such that $\cos(\theta)=n$.

Like [cos](#)(, the result of \cos^{-1} (depends on whether the calculator is in [Radian](#) or [Degree](#) mode. However, unlike cosine, the result is in degrees or radians, not the argument. A full rotation around a circle is 2π radians, which is equal to 360° . The conversion of $\theta=\cos^{-1}(n)$ from radians to degrees is $\theta*180/\pi$ and from degrees to radians is $\theta*\pi/180$. The \cos^{-1} (command also works on a list.

The \cos^{-1} (function can be defined for all real and complex numbers, but assumes real values only in the closed interval $[-1,1]$. Because Z80 calculators have their trigonometric functions and inverses restricted only to real values, the calculator will throw [ERR:DOMAIN](#) if the argument is outside of this interval, no matter what the mode setting may be.

In radians:

```
:cos-1(-1)  
3.141592654
```

```
cos-1(1  
cos-1(.5)/1°  
cos-1((0, √(3))/2, .  
{1.570796327 .5...
```

Command Summary

Returns the inverse cosine (also called arccosine)

Command Syntax

$\cos^{-1}(number)$

Menu Location

Press:

1. [2nd]
2. [\cos^{-1}]

Calculator Compatibility

TI-83/84/+/SE

In degrees:

```
:cos-1(-1)  
180
```

Token Size

1 byte

Advanced Uses

Since the function cosine itself doesn't have the restrictions that arccosine does, and since arccosine is the inverse of cosine, you can use $\cos^{-1}(\cos(\theta))$ to keep a variable within a certain range (most useful for the [home screen](#)). Here is an example for a game like [pong](#). The ball travels between 0 and 12.

You could use a flag like this:

```
:If X=12 or not(X)      \ X is the position  
:-D→D                  \ D is the direction  
:X+D→X                 \ new position  
:Output(8,X,"=
```

An easier way to do this, without needing a flag or even an If statement, is using $\cos^{-1}(\cos(\theta))$

```
:X+1→X                  \ Note: the calculator is in Degree mode  
:Output(8,cos-1(cos(15X))/15,"=")    \ I used 15 because cos-1 ranges  
                                         and X from [0,12], so 180
```

Error Conditions

- **ERR:DOMAIN** is thrown if you supplied an argument outside the interval [-1,1]
- **ERR:DATA TYPE** is thrown if you input a complex value or a matrix.

Related Commands

- [sin\(](#)
- [sin⁻¹\(](#)
- [cos\(](#)
- [tan\(](#)
- [tan⁻¹\(](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/arccos>

The $\cos(\theta)$ Command

$\cos(\theta)$ returns the cosine of θ , which is defined as

the x-value of the point of intersection of the unit circle and a line containing the origin that makes an angle θ with the positive x-axis

The value returned depends on whether the calculator is in Radian or Degree mode. A full rotation around a circle is 2π radians, which is equal to 360° . The conversion from radians to degrees is $\text{angle} * 180/\pi$ and from degrees to radians is $\text{angle} * \pi/180$. The `cos(` command also works on a list of real numbers.

In radians:

```
cos(π/3)  
.5
```

In degrees:

```
cos(60)  
.5
```

Advanced Uses

You can bypass the mode setting by using the $^\circ$ (degree) and r (radian) symbols. These next two commands will return the same values no matter if your calculator is in degrees or radians:

```
cos(60°)  
.5
```

```
cos(π/3^r)  
.5
```

Error Conditions

- ERR:DATA TYPE is thrown if you supply a matrix or a complex argument.

Related Commands

- [sin\(](#)
- [sin⁻¹\(](#)
- [cos⁻¹\(](#)
- [tan\(](#)
- [tan⁻¹\(](#)

See Also

<code>cos(θ)</code>	1
<code>cos(180°)</code>	-1
<code>cos(0, π/3, 180°, 2)</code>	
<code>{1 .5 -1 -.4161...</code>	

Command Summary

Returns the cosine of a real number.

Command Syntax

`cos(angle)`

Menu Location

Press [COS]

Calculator Compatibility

TI-83/84/+SE

Token Size

1 byte

- [Look-Up Tables](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/cos>

The \cosh^{-1} (Command

The \cosh^{-1} (function gives the inverse hyperbolic cosine of a value. $\cosh^{-1}(x)$ is the number y such that $x = \cosh(y)$.

Although $\cosh^{-1}(x)$ can be defined for all real and complex numbers, it assumes real values only for $x \geq 1$. Since hyperbolic functions in the Z80 calculators are restricted only to real values, [ERR:DOMAIN](#) is thrown when $x < 1$.

The \cosh^{-1} (command also works for lists.

```
cosh-1(1)  
0  
cosh-1({2,3})  
{1.316957897 1.762747174}
```

Error Conditions

- [ERR:DOMAIN](#) when taking the inverse cosh of a number less than 1.

Related Commands

- [sinh\(](#)
- [sinh⁻¹\(](#)
- [cosh\(](#)
- [tanh\(](#)
- [tanh⁻¹\(](#)

```
cosh-1(1) 0  
cosh-1(3) 1.762747174  
cosh-1(4E99) 230.0353657
```

Command Summary

Calculates the inverse hyperbolic cosine of a value.

Command Syntax

$\cosh^{-1}(\text{value})$

Menu Location

The \cosh^{-1} (command can only be found in the catalog. Press:

1. 2nd CATALOG to access the command catalog.
2. C to skip to commands starting with C.
3. Scroll down and select \cosh^{-1} (

Calculator Compatibility

TI-83/84/+SE

Token Size

1 byte

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/arcosh>

The \cosh (Command

Calculates the hyperbolic cosine of a value. The hyperbolic trig functions [sinh\(](#), [cosh\(](#), and [tanh\(](#) are an analog of normal trig functions, but for a

```
cosh(0) 1
```

hyperbola, rather than a circle. They can be expressed in terms of real powers of e , and don't depend on the Degree or Radian mode setting.

```
cosh(0)  
1  
cosh(1)  
1.543080635
```

Like normal trig commands, cosh works on lists as well, but not on complex numbers, even though the function is often extended to the complex numbers in mathematics.

Formulas

The definition of hyperbolic cosine is:

$$\cosh x = \frac{e^x + e^{-x}}{2} \quad (1)$$

Related Commands

- [sinh](#)
- [sinh⁻¹](#)
- [cosh⁻¹](#)
- [tanh](#)
- [tanh⁻¹](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/cosh>

The CubicReg Command

The CubicReg command can calculate the best fit cubic function through a set of points. To use it, you must first store the points to two lists: one of the x-coordinates and one of the y-coordinates, ordered so that the Nth element of one list matches up with the Nth element of the other list. L1 and L2 are the default lists to use, and the List Editor (STAT > Edit...) is a useful window for entering the points. You must have at least 4 points, because there's infinitely many cubics that can go through 3 points or less.

In its simplest form, CubicReg takes no arguments, and calculates a cubic through the points in L1 and L2:

```
cosh(1n(2) 1.25  
cosh(230 3.86100925e99
```

Command Summary

Calculates the hyperbolic cosine of a value.

Command Syntax

`cosh(value)`

Menu Location

The `cosh` command is only found in the Catalog. Press:

1. 2nd CATALOG to access the command catalog.
2. C to skip to commands starting with C.
3. Scroll down and select `cosh`.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

```
CubicReg  
y=ax3+bx2+cx+d  
a=1.4  
b=-3.414285714  
c=8.785714286  
d=-6.04  
R2=.9995297853
```

Command Summary

Calculates the best fit cubic function through a set of points.

Command Syntax

```
: {9,13,21,30,31,31,34→L1  
: {260,320,420,530,560,550,590→L2  
: CubicReg
```

On the home screen, or as the last line of a program, this will display the equation of the quadratic: you'll be shown the format, $y=ax^3+bx^2+cx+d$, and the values of a, b, c, and d. It will also be stored in the RegEQ variable, but you won't be able to use this variable in a program — accessing it just pastes the equation wherever your cursor was. Finally, the statistical variables a, b, c, d, and R^2 will be set as well. This latter variable will be displayed only if "Diagnostic Mode" is turned on (see [DiagnosticOn](#) and [DiagnosticOff](#)).

You don't have to do the regression on L1 and L2, but if you don't you'll have to enter the names of the lists after the command. For example:

```
: {9,13,21,30,31,31,34→FAT  
: {260,320,420,530,560,550,590→CALS  
: CubicReg ↵FAT,↵CALS
```

You can attach frequencies to points, for when a point occurs more than once, by supplying an additional argument — the frequency list. This list does not have to contain integer frequencies. If you add a frequency list, you must supply the names of the x-list and y-list as well, even when they're L1 and L2.

Finally, you can enter an equation variable (such as Y_1) after the command, so that the equation is stored to this variable automatically. This doesn't require you to supply the names of the lists, but if you do, the equation variable must come last. You can use polar, parametric, or sequential variables as well, but since the quadratic will be in terms of X anyway, this doesn't make much sense.

An example of CubicReg with all the optional arguments:

```
: {9,13,21,30,31,31,34→FAT  
: {260,320,420,530,560,550,590→CALS  
: {2,1,1,1,2,1,1→FREQ  
: CubicReg ↵FAT,↵CALS,↵FREQ, Y1
```

Advanced

Note that even if a relationship is actually linear or quadratic, since a cubic regression has all the freedom of a linear regression and more, it will produce a better R^2 value, especially if the number of points is small, and may lead you to (falsely) believe that a relationship is cubic when it actually isn't. Take the correlation constant with a grain of salt, and consider if the fit is really that much better at the expense of doubling the complexity, and if there's any reason to believe the relationship between the variables may be cubic.

CubicReg [*x-list, y-list, [frequency list], [equation variable]*]

Menu Location

Press:

1. [STAT] to access the statistics menu
2. [LEFT] to access the CALC submenu
3. 6 to select CubicReg, or use arrows

Calculator Compatibility

TI-83/84/+SE

Token Size

1 byte

Related Commands

- LinReg(ax+b)
- QuadReg
- QuartReg

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/cubicreg>

The cumSum(Command

cumSum(calculates the cumulative sums of a list, or of the columns of a matrix, and outputs them in a new list or matrix variable.

For a list, this means that the Nth element of the result is the sum of the first N elements of the list:

```
cumSum({1,3,5,7,9})  
{1 4 9 16 25}
```

For a matrix, cumSum(is applied to each column in the same way as it would be for a list (but numbers in different columns are never added):

```
[[0,1,1][0,1,3][0,1,5][0,1,7]]  
[[0 1 1]  
 [0 1 3]  
 [0 1 5]  
 [0 1 7]]  
cumSum(Ans)  
[[0 1 1]  
 [0 2 4]  
 [0 3 9]  
 [0 4 16]]
```

Advanced Uses

The ΔList(command is very nearly the inverse of the cumSum(command - it calculates the differences between consecutive elements. For any list, ΔList(cumSum(list)) will return the same list, but without its first element:

```
ΔList(cumSum({1,2,3,4,5,6,7}))  
{2 3 4 5 6 7}
```

Removing the first element would otherwise be a difficult procedure involving the seq(command, so

```
{1,1,1,1,1  
 {1 1 1 1 1}  
cumSum(Ans  
 {1 2 3 4 5}  
cumSum(Ans  
 {1 3 6 10 15}
```

Command Summary

Calculates cumulative sums of a list or of the columns of a matrix.

Command Syntax

`cumSum(list or matrix)`

Menu Location

Press:

1. 2nd LIST to access the list menu.
2. RIGHT to access the OPS submenu.
3. 6 to select cumSum(, or use arrows.

Alternatively, press:

1. MATRIX (TI-83) or 2nd MATRIX (TI-83+ or higher) to access the matrix menu.
2. RIGHT to access the MATH submenu.
3. 0 to select cumSum(, or use arrows.

Calculator Compatibility

TI-83/84/+/SE

Token Size

this is a useful trick to know.

2 bytes

For a matrix, if you want to sum up the rows instead of the columns, use the T (transpose) command.

Related Commands

- Δ List(
- T (transpose)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/cumsum>

The dayOfWk(Command

`dayOfWk(year,month,day)` returns an integer from 1 to 7, each representing a separate day of the week. 1 represents Sunday, 2 represents Monday, and so on, until 7 goes with Saturday. The date format is different than the normal American format (month/day/year), so be careful to put the arguments in the right order.

```
:dayOfWk(2007,12,30)
```

The above code returns 1, because the 30th of December, 2007, is a Sunday.

Error Conditions

- **ERR:DOMAIN** is thrown if any of the arguments are non-integral, or the date does not exist, such as the 42nd of February. However, the year does not matter (a date that takes place in the year 10000 is valid).

Related Commands

- dbd(
- setDate(
- getDate
- getDtFmt

See Also

- Day of Week — routine to calculate the day of the week

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/dayofwk>

```
dayOfWk(2007,12,  
30
```

1

Command Summary

Returns an integer from 1 to 7, each representing a day of the week, given a date.

Command Syntax

`dayOfWk(year,month,day)`

Menu Location

Press:

1. [2ND] + [0] for the CATALOG
2. [X^{-1}] to jump to the letter D
3. [ENTER] to insert the command

Calculator Compatibility

TI-84+/SE

Token Size

2 bytes

The dbd(Command

The dbd(command calculates the number of days between two dates. Each date is encoded as a single number in one of two formats (two formats can be mixed in the same command):

- day, month, year — DDMM.YY (e.g. April 26, 1989 would be 2604.89)
- month, day, year — MM.DDYY (e.g. April 26, 1989 would be 04.2689 or just 4.2689)

Because this is just a number like any other, leading zeroes and trailing zeroes after the decimal can be dropped. For example, January 1, 2000 does not have to be formatted as 0101.00 but can be simply 101.

Since there are only two digits for the year, obviously only a century's worth of dates can be handled. The calculator assumes this range to be from January 1, 1950 to December 31, 2049.

If the second date comes before the first, dbd(will return a negative number of days, so the range of possible results is from -36524 to 36524.

Finally, dbd(will also work for list inputs in the usual manner: a single date will be compared against every date in a list, and two lists of dates will be paired up.

```
dbd(612.07,2512.07
     19
dbd(1.0207,1.0107
     -1
dbd(1.0107,{2.0107,3.0107,4.0107})
     {31 59 90}
```

Advanced Uses

The dbd(command can be used to calculate the day of week without using the dayOfWk(command, which is only available on the TI-84+ and TI-84+ SE.

Error Conditions

- ERR:DOMAIN is thrown if a date is improperly formatted.

dbd(1701.96,1701 .97	366
dbd(01.1796,01.1 797	366

Command Summary

Calculates the number of days between two days.

Command Syntax

dbd(*date1*, *date2*)

Date format — one of:

- DDMM.YY
- MM.DDYY

Menu Location

On the TI-83, press:

1. 2nd FINANCE to access the finance menu.
2. ALPHA D to select dbd(, or use arrows and ENTER.

On the TI-83+ or higher, press:

1. APPS to access the applications menu.
2. ENTER to select Finance...
3. ALPHA D to select dbd(, or use arrows and ENTER.

Calculator Compatibility

TI-83/84+/SE

Token Size

2 bytes

Related Commands

- dayOfWk(

See Also

- Day of Week

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/dbd>

The ► Dec Command

This command is completely useless. Its supposed use is to convert numbers into decimal form, but any typed fractions are displayed as decimals anyway.

```
1/3  
.3333333333  
1/3►Dec  
.3333333333
```

7/2	3.5
7/2►Frac	7/2
7/2►Dec	3.5
█	

Related Commands

- ►Frac

Command Summary

Displays the decimal form of a fraction.

Command Syntax

<fraction>► Dec

Menu Location

While editing a program, press:

1. MATH to enter the MATH menu
- 2 or use the arrow keys to select

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/dec>

The Degree Command

The Degree command puts the calculator into Degree mode, where the inputs and/or outputs to trig functions are assumed to be degree angles.

Angles measured in degrees range from 0 to 360, with 0 being an empty angle, 90 being a right angle, 180 being a straight angle, and 360 being a full angle all the way around a circle.

To convert from a degree angle to a radian angle, multiply by $180/\pi$. To go the other way, and get a radian angle from a degree angle, multiply by $\pi/180$.

The following commands are affected by whether the calculator is in Radian or Degree mode:

The input is differently interpreted:

- P►Rx(, P►Ry(
- sin(, cos(, tan(

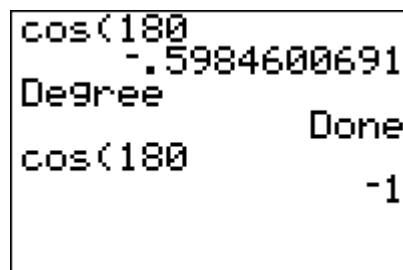
The output is differently expressed:

- angle(
- R►Pθ(
- sin⁻¹(, cos⁻¹(, tan⁻¹(
- ►Polar (and complex numbers when in re^{θi} mode)
- [, °

However, some commands are notably unaffected by angle mode, even though they involve angles, and this may cause confusion. This happens with the SinReg command, which assumes that the calculator is in Radian mode even when it's not. As a result, the regression model it generates will graph incorrectly in Degree mode.

Also, complex numbers in polar form are an endless source of confusion. The angle(command, as well as the polar display format, are affected by angle mode. However, complex exponentials (see the e^{iθ} command), defined as $e^{i\theta} = \cos\theta + i\sin\theta$, are evaluated as though in Radian mode, regardless of the angle mode. This gives mysterious results like the following:

```
Degree:re^θi
      Done
e^(πi)
      1e^(180i)
Ans=e^(180i)
      0 (false)
```



```
cos(180
      -.5984600691
Degree
      Done
cos(180
      -1
```

Command Summary

Puts the calculator in Degree mode.

Command Syntax

Degree

Menu Location

While editing a program, press:

1. MODE to access the mode menu.
2. Use arrows and ENTER to select Degree.

Calculator Compatibility

TI-83/84/+SE

Token Size

1 byte

Overall, it's better to put your calculator in Radian mode when dealing with polar form of complex numbers, especially since no mathematician would ever use degrees for the purpose anyway.

Optimization

It's sometimes beneficial to use the \circ symbol instead of switching to Degree mode. The \circ symbol will make sure a number is interpreted as a degree angle, even in radian mode, so that, for example:

```
Radian
      Done
sin(90)
      - .8011526357
sin(90°)
      1
```

This is smaller when only one trig calculation needs to be done. Also, it doesn't change the user's settings, which are good to preserve whenever possible.

Related Commands

- [Radian](#)
- [\$\text{L}\$](#)
- [\$\circ\$](#)
- [\$\text{-}\$](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/degree-mode>

The \circ (Degree Symbol) Command

Normally, when the calculator is in radian mode, the trigonometric functions only return values calculated in radians. With the \circ symbol you can have the angle evaluated as if in degree mode because it converts the angle into radians.

One full rotation around a circle is 2π radians, which is equal to 360° . To convert an angle in radians to degrees you multiply by $180/\pi$, and to convert from degrees to radians multiply by $\pi/180$.

In radian mode:

```
sin(45)          \ actually calculatin
      .8509035245
sin(45°)
      .7071067812
```

```
sin(30
      -.9880316241
sin(30°
      .5
180°
      3.141592654
```

Command Summary

If the calculator is in radian mode, the \circ (degree) symbol converts an angle to radians.

Command Syntax

angle \circ

Menu Location

Press:

```
sin(45)
.7071067812
sin(45°)
.7071067812 \ There's no diff
```

1. [2nd]
2. [Angle]
3. [Enter] or [1]

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

```
:Degree
:sin(X)
can be
:sin(X°)
```

Related Commands

- r (radian symbol)

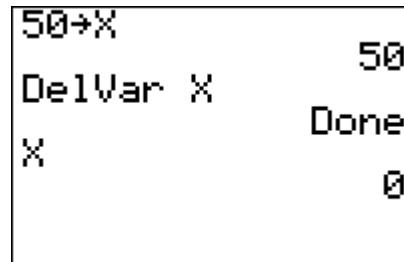
For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/degree-symbol>

The DelVar Command

The DelVar command deletes the contents of a variable (and thus the variable itself) from memory. You can use the DelVar command with any variable: reals, lists, matrices, strings, pictures, etc. However, you cannot use DelVar on specific elements of a matrix or string; it will actually throw a ERR:SYNTAX error. (It also does not work on programs, unfortunately.)

If the DelVar command is used with a real variable, the variable is not only deleted from memory but automatically set to zero the next time it is used. This is equivalent to using store (\rightarrow) to manually set the variable yourself. Because the DelVar command is two bytes instead of one, there is no size difference between the two.

```
:0→A
same as
:DelVar A
```



Command Summary

Deletes a variable from memory.

Command Syntax

DelVar *variable*

Menu Location

While editing a program, press:

1. PRGM to enter the PRGM menu

While there is no size difference between the two, DelVar does have some problems that go along with using it. If used in a For loop to delete the counter variable or used to delete the variable and/or value in the IS>(or DS<(commands before using them, it will cause an ERR:UNDEFINED error.

This is a result of the way that the interpreter in TI-Basic is designed, so there is nothing you can do about it. You just need to be cognizant of it when using DelVar in a For loop or together with IS>(or DS<(.

2. ALPHA TAN to choose DelVar, or use arrows

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

Advanced Uses

When you are done using variables, you should delete them at the end of the program with the DelVar command to cleanup. Each variable takes up a set amount of space (for example, a real variable is 15 bytes), and the more variables you can delete the more free memory is available. Free memory helps your programs run faster and allows you to pack more things on your calculator.

Because the DelVar command doesn't update the Ans variable, you can use DelVar and the current value in Ans will still be preserved for later use.

Optimizations

The DelVar command does not need a line break or colon (which indicates a new line of code) following the variable name. This allows you to make chains of variables (organized in whatever order you want), and it saves a byte for each line break or colon removed.

```
:DelVar A  
:DelVar B  
can be  
:DelVar ADelVar B
```

Besides making chains of variables, the DelVar command also allows you to take the command from the next line and put it immediately after the DelVar command.

```
:DelVar A  
:Disp "Hello  
can be  
:DelVar ADisp "Hello
```

There are, however, two cases in which the following statement will be ignored, so you should add a newline:

- The End from an If statement.
- A Lbl command.

Command Timings

The speed of the DelVar command depends on the circumstance where it is used. When the

variable already exists, DelVar is slower because it has to perform garbage collection. When the variable does not exist, however, its speed is greatly increased because it does not have to do anything to setup the variable.

Error Conditions

- **ERR:SYNTAX** is thrown when trying to delete a system variable (e.g. DelVar Xmin) or a program, even though this is syntactically correct.
- **ERR:UNDEFINED** is thrown if you delete the loop variable while inside the loop, or delete the variable used in IS>() or DS<().

Related Commands

- ClrList

See Also

- Program Cleanup

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/delvar>

The DependAsk Command

When the DependAsk setting (opposed to the DependAuto setting) is turned on, values in the table are not automatically calculated. To calculate the value of an equation, you have to select the column corresponding to that equation in the row corresponding to the value at which to calculate it, and press ENTER. For example, to calculate Y1 at X=0, select the X=0 column, scroll right to Y1, and press ENTER.

The DependAsk setting might be useful when dealing with a difficult-to-calculate function, for which you wouldn't want to have to calculate values that aren't really necessary.

Related Commands

- IndpntAuto
- IndpntAsk
- DependAuto
- DispTable



Command Summary

Disables automatic calculations in the table.

Command Syntax

DependAsk

Menu Location

Press:

1. 2nd TBLSET to access the table settings menu.
2. Use arrows and ENTER to select Ask from the Depend: line.

Calculator Compatibility

Token Size

1 byte

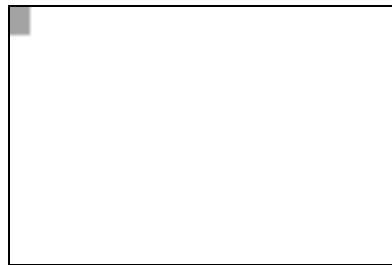
For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/dependask>

The DependAuto Command

When the DependAuto setting (opposed to the DependAsk setting) is turned on, values in the table are automatically calculated. With IndpntAuto, that means the table is automatically filled out completely; with IndpntAsk, that means that as soon as you enter a value for the independent variable, all the values of the dependent variables are calculated. This is usually the setting you want to use.

Related Commands

- IndpntAuto
- IndpntAsk
- DependAsk
- DispTable



Command Summary

Enables automatic calculations in the table.

Command Syntax

DependAuto

Menu Location

Press:

1. 2nd TBLSET to access the table settings menu.
2. Use arrows and ENTER to select Auto from the Depend: line.

Calculator Compatibility

TI-83/84+/SE

Token Size

1 byte

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/dependauto>

The det(Command

The `det(` command calculates the determinant of a square matrix. If its argument is not a square matrix, ERR:INVALID DIM will be thrown.

Advanced Uses

If $[A]$ is an $N \times N$ matrix, then the roots of $\det([A] - X \text{identity}(N))$ are the eigenvalues of $[A]$.

Formulas

For 2×2 matrices, the determinant is simply

$$\det([a \ b \ d]) = |a \ b \ d| = ad - bc \quad (1)$$

For larger matrices, the determinant can be computed using the Laplace expansion, which allows you to express the determinant of an $n \times n$ matrix in terms of the determinants of $(n-1) \times (n-1)$ matrices. However, since the Laplace expansion takes $O(n!)$ operations, the method usually used in calculators is Gaussian elimination, which only needs $O(n^3)$ operations.

The matrix is first decomposed into a unit lower-triangular matrix and an upper-triangular matrix using elementary row operations:

(2)

$$(1 \ \ \ \vdots \ \ddots \ \times \ \dots \ 1) (\times \ \dots \ \times \ \ddots \ \vdots \ \times)$$

The determinant is then calculated as the product of the diagonal elements of the upper-triangular matrix.

Error Conditions

- ERR:INVALID DIM is thrown when the matrix is not square.

Related Commands

- `identity(`
- `ref(`
- `rref(`

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/det>

```
[[1,5,5][2,0,4][3,2,3]
 [[1 5 5]
 [2 0 4]
 [3 2 3]]
det(Ans) 42
```

Command Summary

Calculates the determinant of a square matrix.

Command Syntax

`det(matrix)`

Menu Location

Press:

1. MATRIX (83) or 2nd MATRIX (83+ or higher) to access the matrix menu
2. LEFT to access the MATH submenu
3. ENTER to select `det()`.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

The DiagnosticOff Command

After the DiagnosticOff command is executed, all regression commands found in the STAT>CALC menu, as well as LinRegTTest, will not display the correlation statistics r and r^2 (or just R^2 in some cases). This is already turned off by default, although there is no disadvantage whatsoever to turning it on. To reverse this command, execute the DiagnosticOn command.

The statistic r , known as the correlation coefficient, measures the strength and direction of any linear relationship in the data (therefore if your regression model isn't linear, it may not exist, unless the calculator performed a transformation on the data). If r is close to 1, then the relationship is strong, and positive (that is, the variables increase and decrease together). If r is close to -1, then the relationship is strong, and negative (that is, as one variable increases, the other decreases). If r is close to 0, there is no linear relationship.

The statistic r^2 or R^2 is equal to the square of the above value (when it exists), and is also a measure of the strength of a relationship. Specifically, it represents the proportion of variance in the dependent variable that is accounted for by the regression model. If this value is close to 1, there is a strong relationship; if it's close to 0, there is either no relationship or the regression model doesn't fit the data.

Advanced

Although these statistics are a good indication of whether a regression curve is good or not, they are not infallible. For example, the initial portion of data that actually correlates exponentially may well appear linear, and have a high correlation coefficient with a linear fit.

Another good way to check a regression curve is to look at the plot of the residuals vs. the x-values. If the regression curve is a good fit, then this plot should appear random in going from positive to negative. However, should you see a distinct pattern - say, if you tried a linear fit but the residual plot looks vaguely parabolic - you know you should try a different regression curve.

You should also consider what your regression line implies about the nature of the data, and vice versa. For example, if you're comparing the height of release of a ball to the time it takes to fall, a natural assumption is that the regression curve should pass through $(0,0)$, and a curve that doesn't do that may be incorrect. However, take this advice with a grain of salt: if your curve fits the data points you put in but not such natural-assumption points, that may simply



Command Summary

Changes settings so that the correlation variables, r and r^2 , are not displayed when calculating a regression

Command Syntax

DiagnosticOff

Menu Location

Press:

1. 2ND CATALOG to access the command catalog
2. D to skip to commands starting with D
3. Scroll down and select DiagnosticOff

(The DiagnosticOff command can't be found outside the catalog)

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

mean that the curve works on a limited domain. Or, it may mean your assumptions are wrong.

Command Timings

Although the correlation statistics are not displayed with DiagnosticOff, they are calculated in either case. This means that DiagnosticOn and DiagnosticOff will not change how fast regressions are calculated.

Related Commands

- DiagnosticOn

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/diagnosticoff>

The DiagnosticOn Command

After the DiagnosticOn command is executed, all regression commands found in the STAT>CALC menu, as well as LinRegTTest, will display the correlation statistics r and r^2 (or just R^2 in some cases). This is turned off by default, but there is no disadvantage whatsoever to turning it on. To reverse this command, execute the DiagnosticOff command.

The statistic r , known as the correlation coefficient, measures the strength and direction of any linear relationship in the data (therefore if your regression model isn't linear, it may not exist). If r is close to 1, then the relationship is strong, and positive (that is, the variables increase and decrease together). If r is close to -1, then the relationship is strong, and negative (that is, as one variable increases, the other decreases). If r is close to 0, there is no linear relationship.

The statistic r^2 or R^2 is equal to the square of the above value (when it exists), and is also a measure of the strength of a relationship. Specifically, it represents the proportion of variance in the dependent variable that is accounted for by the regression model. If this value is close to 1, there is a strong relationship; if it's close to 0, there is either no relationship or the regression model doesn't fit the data.

Advanced

Although these statistics are a good indication of whether a regression curve is good or not, they are not infallible. For example, the initial portion of data that actually correlates exponentially may well appear linear, and have a high correlation coefficient



Command Summary

Changes settings so that the correlation variables, r and r^2 , are displayed when calculating a regression

Command Syntax

DiagnosticOn

Menu Location

Press:

1. 2ND CATALOG to access the command catalog
2. D to skip to commands starting with D
3. Scroll down and select DiagnosticOn

(The DiagnosticOn command can't be found outside the catalog)

Calculator Compatibility

TI-83/84/+/SE

with a linear fit.

Another good way to check a regression curve is to look at the plot of the residuals vs. the x-values. If the regression curve is a good fit, then this plot should appear random in going from positive to negative. However, should you see a distinct pattern - say, if you tried a linear fit but the residual plot looks vaguely parabolic - you know you should try a different regression curve.

You should also consider what your regression line implies about the nature of the data, and vice versa. For example, if you're comparing the height of release of a ball to the time it takes to fall, a natural assumption is that the regression curve should pass through (0,0), and a curve that doesn't do that may be incorrect. However, take this advice with a grain of salt: if your curve fits the data points you put in but not such natural-assumption points, that may simply mean that the curve works on a limited domain. Or, it may mean your assumptions are wrong.

Command Timings

Although the correlation statistics are displayed with DiagnosticOn, they are calculated in either case. This means that DiagnosticOn and DiagnosticOff will not change how fast regressions are calculated.

Related Commands

- DiagnosticOff

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/diagnosticon>

The dim(Command

The dim(command is used to find the size of a list or matrix. It takes only one argument - the list or matrix you want the size of. For a list, it returns the number of elements; for a matrix, it returns a two-element list of the number of rows and the number of columns.

```
:dim(L1  
5  
:dim([A]  
{2,3}
```

The dim(command can also be used to change the size of a list or matrix; this is perhaps its most important use. To do this, just store the desired size to the list or matrix (the dim(command is the only one you can store to as though it were a variable).

```
:7→dim(L1  
{2,2→dim([A]
```

Token Size

2 bytes

```
dim(L1  
5→dim(L1  
L1  
{1 2 3 0 03
```

Command Summary

Returns the size of a list or matrix. Can also be used to change the size.

Command Syntax

dim(*list*)

dim(*matrix*)

length→dim(*list*)

{rows,columns→dim(*matrix*)}

For a list, if this increases the size, zero elements will be added to the end of the list; if this decreases the size, elements will be removed starting from the end.

For a matrix, if this increases the number of rows or columns, new rows or columns filled with zeros will be added to the bottom and right respectively. If this decreases the number of rows and columns, those rows and columns will be removed starting from the bottom (for rows) and right (for columns).

If a list or matrix doesn't exist before its size is changed, the dim(command will actually create it with the correct size. All the elements, in this case, will be set to 0.

Advanced Uses

In the case of lists, the dim(command is used in adding an element to the end of a list. Although augment(can be used for the same task, dim(is faster - but takes more memory. For example, to add the element 5 to the end of L1:

```
: 5→L1(1+dim(L1)
```

It's also possible, using the dim(command, to set the size of a list to 0. In this case, the list exists, but doesn't take up any memory, and cannot be used in expressions (similar to the output of ClrList). This is not really useful.

Optimization

When creating a list or matrix using dim(, all the elements are preset to 0; this can be used in place of the Fill(command to set a list or matrix to a bunch of zeros in a program. Since we don't usually know for sure that the list or matrix doesn't exist, we must first delete it with DelVar.

```
: {5,5→dim([A]  
: Fill(0,[A]  
can be  
: DelVar [A] {5,5→dim([A])
```

Menu Location

Press:

1. 2nd LIST to access the list menu
2. RIGHT to access the OPS submenu
3. 3 to choose dim(, or use arrows

Alternatively, press:

1. MATRIX (83) or 2nd MATRIX (83+ or higher) to access the matrix menu
2. RIGHT to access the MATH submenu
3. 3 to choose dim(, or use arrows

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

Error Conditions

- **ERR:INVALID DIM** is thrown if you try to make a list or matrix bigger than 999 or 99x99 elements respectively, or if you try to create a matrix that isn't 2-dimensional.

Related Commands

- length
- Fill
- augment

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/dim>

The Disp Command

The first, and easiest, way to display text is using the Disp command. You can display whatever combination of text and values that you want. Text is displayed on the left side of the screen, while numbers, variables and expressions are displayed on the right side. Text can be moved over to the right by padding it with spaces, but there is no equivalent for numbers, variables, and expressions.

When displaying a matrix or a list, and the matrix or list is too large to display in its entirety, an ellipsis (...) is displayed at the boundaries of the screen. The matrix or list, unfortunately, cannot be scrolled so the rest of it can be seen (use the Pause command instead).

With the small screen size, you have to keep formatting in mind when displaying text. Because the text does not wrap to the next line if it is longer than sixteen characters, the text gets cut off and an ellipsis is displayed at the end of the line. When the text you want to display is longer than sixteen characters, you should break the text up and display each part with its own Disp command.

```
:Disp "Just Saying Hello
Break the text up
:Disp "Just Saying
:Disp "Hello
```

The Disp command displays text line by line, giving each argument its own blank line. If the screen is clear, the arguments are displayed beginning at the first line. But if there is text on the first line, the arguments are displayed beginning at the first available blank line. When all the lines have text on them including the last, the screen will automatically scroll up until every line is blank.

This means that, while a Disp command can technically display an unlimited amount of lines of text, you should not display more than seven consecutive lines of text at any one time (because of the screen height). If there are too many arguments, the arguments that were displayed will be pushed up out of sight, to allow the other arguments to be displayed. This is usually not desired, but it can be used to create some cool scrolling effects by messing with the text that you display.

PROGRAM: EXAMPLE
:Disp "AN EXAMPL
E", "BY TIBASICDE
V

Command Summary

Displays an expression, a string, or several expressions and strings on the home screen.

Command Syntax

Disp [argument1,argument2,...]

Menu Location

While editing a program, press:

1. PRGM to enter the PRGM menu
2. RIGHT to enter the I/O menu
3. 3 to select Disp (or use arrows to select)

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

TI-Nspire CX CAS

The result is that you can never display text on the last line of the screen using the Disp command; you need to use the Output(command. (Using Output(does not have any affect on Disp and its text.) Also, if you have more than seven lines of text to display, you will need to place the Pause command after every seven lines to prevent the screen from scrolling. These two scenarios come up fairly often, so it is good to know how to deal with them.

```
PROGRAM:DISP
:ClrHome
:Disp A,B,C,D,E,F,G
:Pause
:Disp A,B,C,D,E,F,G
:Output(8,16,H
```

Like other text display commands, you can display each function and command as text. However, this is not without problems as each function and command is counted as one character. The two characters that you can't display are quotation marks ("") and the store command (→). However, you can mimic these respectively by using two apostrophes (''), and two subtract signs and a greater than sign (→).

Advanced Uses

You can use the Disp command by itself, which simply displays the home screen.

```
:Disp
```

When you use an empty string with no text (i.e., two quotes side by side — ""), a blank line is displayed.

```
:Disp ""
```

Optimization

When you have a list of Disp commands (and each one has its own argument), you can just use the first Disp command and combine the rest of the other Disp commands with it. You remove the Disp commands and combine the arguments, separating each argument with a comma. The arguments can be composed of whatever combination of text, numbers, variables, or expressions is desired.

The advantages of combining Disp commands are that it makes scrolling through code faster, and it is smaller when just displaying numbers, variables, or expressions. The disadvantages are that it can hinder readability (make the code harder to read) when you have lots of varied arguments, and it is easier to accidentally erase a Disp command with multiple arguments.

```
:Disp A
:Disp B
Combine the Disp commands
:Disp A,B
```

If you have a string of numbers that you are displaying, you do not need to put quotes around

the numbers. This causes the numbers to be displayed on the right side of the screen, and they cease being a string. You may want to keep the numbers in a string, though, if they have any leading zeros. Because the numbers are no longer in a string, the leading zeros are truncated (taken off).

```
:Disp "2345  
Remove the Quotes  
:Disp 2345
```

Error Conditions

- [ERR:INVALID](#) occurs if this statement is used outside a program.

Related Commands

- [Output\(](#)
- [Text\(](#)
- [Pause](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/Disp>

The DispGraph Command

The DispGraph command displays the graph screen, along with everything drawn or graphed on it.

In many cases, this doesn't need to be done explicitly: commands from the 2nd DRAW menu, as well as many other graph screen commands, will display the graph screen automatically when they are used. Mainly, it's used for displaying the graphs of equations or plots in a program — you would define the variable in question, then use DispGraph to graph it. For example:

```
: "sin(X)"→Y1  
:DispGraph
```

Advanced Uses

DispGraph can also be used to update the graph screen, even if it's already being displayed. For example, changing the value of a plot or equation variable doesn't update the graph immediately. Consider this program:

```
:θ→I
```

PROGRAM: EXAMPLE
: $X^2-X\rightarrow Y_1$
:DispGraph

Command Summary

Displays the graph screen.

Command Syntax

DispGraph

Menu Location

While editing a program, press:

1. PRGM to access the program menu.
2. RIGHT to access the I/O submenu.
3. 4 to select DispGraph, or use arrows and ENTER.

Calculator Compatibility

```
: "I sin(X)" → Y1
: DispGraph
: For(I, 1, 10)
: End
```

At first, it graphs the equation $Y=I\sin(X)$ with $I=0$. Since after that, I is cycled from 1 to 10, and we are on the graph screen, it would be reasonable if this made the program display the graphs of $Y=1\sin(X)$ through $Y=10\sin(X)$. Unfortunately, this isn't the case: though the parameter I changes, the graph screen isn't updated. If, on the other hand, we change the program:

```
: 0 → I
: "I sin(X)" → Y1
: DispGraph
: For(I, 1, 10)
: DispGraph
: End
```

Now the `DispGraph` inside the loop ensures that the graph screen is updated every time, and the program will correctly display all eleven graphs.

Error Conditions

- **ERR:INVALID** occurs if this statement is used outside a program.

Related Commands

- [Disp](#)
- [DispTable](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/dispgraph>

The DispTable Command

This command displays the table screen you normally see by pressing 2nd TABLE, from a running program. The user will see the table screen with a "paused" run indicator, and will be able to use arrows to scroll through it. Pressing ENTER will exit the screen and continue the program.

Advanced Uses

The user can't select any cells in the table to be evaluated if they're not, already. So it's best to select the `IndpntAuto` and `DependAuto` options from the 2nd TBLSET menu before using this command. `IndpntAsk` can also work, however, as long as you store to `TblInput` first.

TI-83/84/+/SE

Token Size

1 byte

PROGRAM: EXAMPLE
: $X^2-X \rightarrow Y_1$
: $\{-2, -1, 0, 1, 2, 3\} \rightarrow$
`TblInput`
`:DispTable`

Command Summary

Displays the table screen.

Command Syntax

`DispTable`

Error Conditions

- **ERR:INVALID** occurs if this statement is used outside a program.

Related Commands

- [IndpntAsk](#)
- [IndpntAuto](#)
- [DependAsk](#)
- [DependAuto](#)

Menu Location

While editing a program, press:

1. PRGM to access the program menu
2. RIGHT to select the I/O submenu
3. 5 to select DispTable, or use arrows and ENTER

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/disptable>

The ►DMS Command

The ►DMS command can be used when displaying a real number on the home screen, or with the [Disp](#) and [Pause](#) commands. It will then format the number as an angle with degree, minute, and second parts.

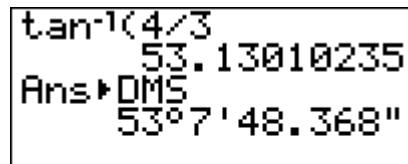
```
30►DMS  
30°0'0"  
100/9°►DMS  
11°6'40"
```

It will also work when displaying a number by putting it on the last line of a program by itself. It does **not** work with [Output\(](#), [Text\(](#), or any other more complicated display commands.

Although the ►DMS is meant as a way to format angles in [Degree mode](#), it doesn't depend on the angle mode chosen, only on the number itself. Note that entering a number as *degree°minute'second"* will also work, in any mode, and it will not be converted to radians in [Radian mode](#).

Error Conditions

- **ERR:SYNTAX** is thrown if the command is used somewhere other than the allowed display commands.



tan⁻¹(4/3)
53.13010235
Ans ►DMS
53°7'48.368"

Command Summary

Formats a displayed number as a degree-minute-second angle.

Command Syntax

value ►DMS

Menu Location

Press:

1. 2nd ANGLE to access the angle menu.
2. 4 to select ►DMS, or use arrows and ENTER.

Calculator Compatibility

TI-83/84/+/SE

- **ERR:DATA TYPE** is thrown if the value is complex, or if given a list or matrix as argument.

Token Size

1 byte

Related Commands

- [►Dec](#)
- [►Frac](#)
- [►Polar](#)
- [►Rect](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/dms>

The Dot Command

The Dot command sets all equations to use the disconnected "dotted-line" graph style: it calculates and draws points on the graph, but doesn't connect them. In addition, this graph style is made the default, so that when a variable is deleted it will revert to this graph style. The other possible setting for this option is Connected.

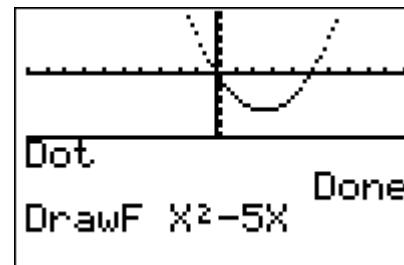
Compare this to the GraphStyle(command, which puts a single equation into a specified graph style.

The Connected and Dot commands don't depend on graphing mode, and will always affect all functions, even in other graphing modes. The exception to this is that sequence mode's default is always the dotted-line style, even when Connected mode is set. The Connected command will still change their graphing style, it just won't change the default they revert to.

In addition to graphing equations, this setting also affects the output of DrawF, DrawInv, and Tangent(.

Advanced Uses

Functions graphed using the dotted-line graph style are very strongly affected by the Xres setting (which determines how many points on a graph are chosen to be graphed). If Xres is a high setting (which means many pixels are skipped), functions in dotted-line mode will be made up of fewer points (in connected mode, this will also be the case, but because the points are connected this isn't as noticeable). You should probably set Xres to 1 if you're going to be using the dotted-line graph style — even 2 is pushing it.



Command Summary

Sets all equations to use the dotted-line graphing style, and makes it the default setting.

Command Syntax

Dot

Menu Location

Press:

1. MODE to access the mode menu.
2. Use arrows to select Dot.

Calculator Compatibility

TI-83/84/+SE

Token Size

2 bytes

Related Commands

- [Connected](#)
- [GraphStyle\(](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/dot>

The DrawF Command

The DrawF commands draws a single expression on the graph screen in terms of X using [Function](#) graphing mode, irregardless of what graphing mode the calculator is actually in. For example, DrawF X² will draw a [parabola](#) in the shape of a U on the screen. Of course, how it is displayed all depends on the window dimensions of the graph screen; you should use a [friendly window](#) to ensure it shows up as you intend.

Advanced Uses

DrawF will update X and Y for each coordinate drawn (like [Tangent\(](#) and [DrawInv](#)), and exit with the last coordinate still stored.

When evaluating the expression using DrawF, the calculator will ignore the following errors:

[ERR:DATA TYPE](#), [ERR:DIVIDE BY 0](#),
[ERR:DOMAIN](#), [ERR:INCREMENT](#), [ERR:NONREAL](#)
[ANS](#), [ERR:OVERFLOW](#), and [ERR:SINGULAR](#)
[MAT](#). If one of these errors occurs, the data point will be omitted.

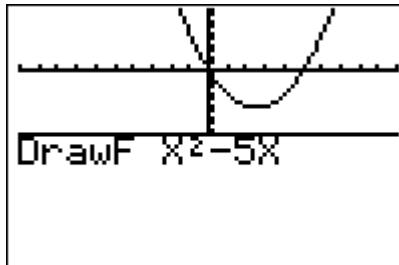
For this reason, DrawF can sometimes behave in an unexpected fashion: for example, it doesn't throw an error for list or matrix expressions (it won't graph anything, either).

You can use DrawF to draw an expression instead of having to store an expression to a Y# variable and then displaying it. At the same time, if you plan on manipulating the expression (either changing the value or changing the expression itself), it would be better to simply use the Y# variable.

Related Commands

- [DrawInv](#)
- [Tangent\(](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/drawf>



Command Summary

Draws an expression in terms of X.

Command Syntax

DrawF *expression*

Menu Location

Press:

1. 2nd PRGM to access the draw menu.
2. 6 to select DrawF, or use arrows and ENTER.

Calculator Compatibility

TI-83/84/+SE

Token Size

1 byte

The DrawInv Command

The DrawInv command draws the inverse of a curve in terms of X. Its single argument is an expression in terms of X.

For example, `DrawInv X2` will draw the inverse of the equation $Y=X^2$. The inverse reverses the variables X and Y, so that the curve $X=Y^2$ will be graphed.

You can also think of this as graphing the expression but with X representing the vertical direction, and Y representing the horizontal.

DrawInv requires the calculator to be in Func mode, and is affected by the Connected/Dot setting.

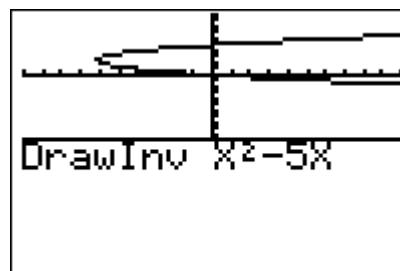
Advanced Uses

DrawInv will update X and Y for each coordinate drawn (like DrawF and Tangent()), and exit with the last coordinate still stored.

When evaluating the expression using DrawInv, the calculator will ignore the following errors:

ERR:DATA TYPE, ERR:DIVIDE BY 0, ERR:DOMAIN, ERR:INCREMENT, ERR:NONREAL ANS, ERR:OVERFLOW, and ERR:SINGULAR MAT. If one of these errors occurs, the data point will be omitted.

For this reason, DrawInv can sometimes behave in an unexpected fashion: for example, it doesn't throw an error for list or matrix expressions (it won't graph anything, either).



Command Summary

Draws the inverse of a curve in terms of X.

Command Syntax

`DrawInv curve`

Menu Location

Press:

1. 2nd DRAW to access the draw menu.
2. 8 to select DrawInv, or use arrows and ENTER.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

Error Conditions

- ERR:MODE is thrown if the calculator is not in Func mode when using DrawInv.

Related Commands

- DrawF
- Tangent(

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/drawinv>

The DS< Command

The decrement and skip if less than command — DS< — is a specialized conditional command. It is equivalent to an If conditional, except the next command will be skipped when the condition is true and it has a variable update built-in. However, it is not used very often (if anything, it is often misused as a looping command) because of its obscure name and somewhat limited application.

The DS< command takes two arguments:

- A variable, which is limited only to one of the real variables (A-Z or θ).
- A value, which can be either a number, variable, or expression (a combination of numbers and variables).

When DS< is executed it subtracts one from the variable (decrements it by one), and compares it to the value. The next command will be skipped if the variable is less than the value, while the next command will be executed if the variable is greater than or equal to the value.

The command DS<(A,B is equivalent to the following code:

```
: A - 1 → A  
: If A ≥ B
```

Here are the two main cases where the DS< command is used:

```
: 5 → A  
: DS<(A, 6  
: Disp "Skipped
```

- Initializes the A variable to 5 and then compares to the value
- $5 < 6$ is true so the display message won't be displayed

```
: 3 → B  
: DS<(B, 2  
: Disp "Not Skipped
```

- Initializes the B variable to 3 and then compares to the value
- $3 < 2$ is false so the display message will be displayed

Note: In addition to both of these cases, there is also the case where the variable and the value are equal to each other. This case is shown below under the 'Advanced Uses' section because it has some added background that goes with it.

PROGRAM: EXAMPLE

```
: 6 → A  
: While A  
: DS<(A, 3  
: Disp A  
: End
```

Command Summary

Decrements a variable by 1 and skips the next command if the variable is less than the value.

Command Syntax

DS<(variable,value)
command

Menu Location

While editing a program, press:

1. PRGM to enter the PRGM menu
2. ALPHA MATH (or 'B') to choose DS<, or use arrows

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

Advanced Uses

When you want the skipping feature of the DS< command to always occur, you just have to use the same variable for both the variable and value arguments of the command:

```
: DS<(B, B
```

An undefined error will occur if the variable and/or value doesn't exist before the DS< command is used, which happens when the DelVar command is used. Consequently, you should not use DelVar with DS<.

A similar code can be used as a substitute for B-1→B if you don't want to change Ans:

```
: DS<(B, B:
```

Note that due to the colon after the line, there will be no statement skipped, so you don't have to worry about that.

Optimization

Because the DS< command has the variable update built-in, it is smaller than manually incrementing a variable by one along with using an If conditional.

```
: A - 1 → A  
can be  
: DS<(A, 0
```

The one caution about this is that if the variable is less than the value (in this case, '0'), the next command will be skipped. If you don't want the skipping functionality, then you need to make sure that the value is never greater than the variable. This is not always possible to do. Also, DS< is slower than its more normal counterpart.

Related to the example code given, DS< should always have a command following after it (i.e. it's not the last command in a program) because it will return an error otherwise. If you have no particular code choice, just put an empty string or something meaningless.

Command Timings

Using DS< to decrement a variable is approximately 25% slower than using code like X-1→X. However, it is faster to use DS< than to construct an If statement to do the same thing.

Note, however, that a quirk in the For(command (see its Optimizations section) will slow down the DS< command significantly if a closing parenthesis is not used for the For(statement.

Error Conditions

- **ERR:INVALID** occurs if this statement is used outside a program.
- **ERR:SYNTAX** is thrown if there is no next line to skip, or if there is only one next line and it is empty.
- **ERR:UNDEFINED** is thrown if the variable to be decremented is not defined.

Related Commands

- IS>(
- If

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/ds>

The e^{\wedge} Command

The e^{\wedge} command raises the constant e to a power. Since it's possible to just type out e , \wedge , and $($, the reason for having a separate function isn't immediately obvious, but in fact most of the time you need to use e , it's to raise it to a power.

The trigonometric and hyperbolic functions can be expressed, and in fact are usually defined, in terms of e^{\wedge} .

e^{\wedge} accepts numbers and lists (but unfortunately not matrices) as arguments. It also works, and is often used for, complex numbers (in fact, one of the standard forms of complex numbers on TI-83 series calculators is $re^{\wedge}\theta i$, which uses the e^{\wedge} function)

```
e^(2)          7.389056099
e^(πi)         -1
e^{(-1,0,1)}   {.3678794412 1 2.718281828}
```

```
e^(0)          1
e^(1)          2.718281828
e^{(-1,0,1,2)} {.3678794412 1 ...}
```

Command Summary

Raises the constant e to the *value* power.

Command Syntax

$e^{\wedge}(value)$

Menu Location

Press [2nd] [e^x] to paste $e^{\wedge}($.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

Formulas

The e^{\wedge} is usually defined by an infinite series:

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} \quad (1)$$

This is then used to define exponentiation in general (for all real and even complex numbers), rather than using some sort of definition of exponents that involves multiplying a number by itself many times (which only works for integers).

Related Commands

- e
- \ln
- 10^{\wedge}

The e Command

e is a constant on the TI-83 series calculators that holds the approximate value of Euler's number, fairly important in calculus and other higher-level mathematics.

The approximate value, to as many digits as stored in the calculator, is 2.718281828459...

The main use of e is as the base of the exponential function e^((which is also a separate function on the calculator), and its inverse, the natural logarithm ln(. From these functions, others such as the trigonometric functions (e.g. sin()) and the hyperbolic functions (e.g. sinh()) can be defined. In re^θi mode, e is used in an alternate form of expressing complex numbers.

Important as the number e is, nine times out of ten you won't need the constant itself when using your calculator, but rather the e^(and ln(functions.

Related Commands

- e^(
- ln(
- log(

A screenshot of a TI-83 calculator screen. It displays two lines of text:
e 2.718281828
π 3.141592654

Command Summary

The mathematical constant e.

Command Syntax

e

Menu Location

Press 2nd e to paste e.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

The E Command

NOTE: Due to the limitations of the wiki markup language, the E command on this page does not appear as it would on the calculator. See Wiki Markup Limitations for more information.

The E symbol is used for entering numbers in scientific notation: it's short for *10^. This means that in many cases, its function is identical to that of the 10^(command (aside from the parenthesis). However, the exponent of E is limited to constant

A screenshot of a TI-83 calculator screen. It displays several lines of text:
1 E6 1000000
{1, 2, 3} E3 {1000 2000 3000}
E -99 1 E -99

Command Summary

The E symbol is used for entering numbers in scientific notation.

integer values -99 to 99.

The E symbol is used in display by the calculator for large numbers, or when in Sci (scientific) or Eng (engineering) mode.

Unlike the exponent of E , the mantissa (a special term for the A in $A \cdot 10^B$, in scientific notation) isn't limited in variable type: it can be a constant, a real or complex variable or expression, a list, a matrix, or even omitted entirely (and then it will be assumed to equal 1). The reason for this versatility is simple: internally, only the exponent is taken to be an actual argument for this command. The rest of the calculation is done through implied multiplication.

```
5E3  
5000  
E -5  
.00001
```

Numbers in scientific notation.

Command Syntax

mantissa E exponent

Menu Location

Press [2nd][EE] to paste the E command.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

Advanced Uses

$\text{E}99$ and $-\text{E}99$ are often used for negative and positive infinity because the TI-83 series of calculators doesn't have an infinity symbol. Commands that often need to use infinity include solve(, fncInt(, normalcdf((and the other distributions), and many others. The error introduced in this way is usually irrelevant, because it's less than the minimum calculator precision, anyway: $\text{E}99$ is mindbogglingly huge.

Optimization

Don't add the mantissa when it's 1: $1\text{E}5$ can just be $\text{E}5$.

In addition, $\text{E}2$ or $\text{E}3$ can be used as shorthand ways of writing 100 and 1000 respectively. This could be continued, in theory, for higher powers of 10, but those aren't necessary as often.

Command Timings

E is much faster than using the 10^{\wedge} command or typing out 10^{\wedge} . The drawback, of course, is that it's limited to constant values.

Related Commands

- $\frac{\wedge}{10}$
- 10^{\wedge}
- e^{\wedge}

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/e-ten>

The ► Eff(Command

The ►Eff(command converts from a nominal interest rate to an effective interest rate. In other words, it converts an interest rate that does not take into account compounding periods into one that does. The two arguments are 1) the interest rate and 2) the number of compounding periods.

For example, take an interest rate of 7.5% per year, compounded monthly. You can use ►Eff(to find out the actual percent of interest per year:

```
►Eff(7.5,12)
7.663259886
```

Formulas

The formula for converting from a nominal rate to an effective rate is:

$$\text{Eff} = 100 \left(\left(1 + \frac{\text{Nom}}{100 \text{ CP}} \right)^{\text{CP}} - 1 \right) \quad (1)$$

Here, Eff is the effective rate, Nom is the nominal rate, and CP is the number of compounding periods.

Error Conditions

- **ERR:DOMAIN** is thrown if the number of compounding periods is not positive, or if the nominal rate is -100% or lower (an exception's made for the nominal rate if there is only one compounding period, since ►Eff(X,1)=X)

Related Commands

- [►Nom\(](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/eff>

The End Command

The End command is used together with the different control structures, including the If conditional, While loop, Repeat loop, and For(loop, to indicate the end of the code block for the respective control structure. In the case of the If conditional, you also need to add a Then command, which is used to indicate the beginning of the control

```
►Eff(10,1
10
►Eff(10,12
10.47130674
►Eff(100,e99
171.8281828
```

Command Summary

Converts a nominal interest rate to an effective interest rate.

Command Syntax

►Eff(*interest rate,compounding periods*)

Menu Location

On the TI-83, press:

1. 2nd FINANCE to access the finance menu.
2. ALPHA C to select ►Eff(.

On the TI-83+ or higher, press:

1. APPS to access the applications menu.
2. ENTER or 1 to select Finance...
3. ALPHA C to select ►Eff(.

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

```
PROGRAM:EXAMPLE
:For(A,1,9,2
:Disp A
:End
```

structure.

Advanced Uses

You can prematurely end a control structure by using a lone If conditional, and then having End be its executed command. Because the calculator stores the positions of the End commands, it will take this End command to be the End command of the control structure.

```
:If <condition>
:End
```

One of the most important features of the End command is putting multiple control structures inside of each other (known as nesting). While you typically nest If conditionals inside of loops, you can actually nest any control structure inside of any other control structure — this even works when using the same control structure, such as a While loop inside of another While loop.

When nesting control structures, you need to remember to put the appropriate number of End commands to close the appropriate structure. The easiest way to keep track of lots of nested control structures is to code the first part, add an End immediately after the beginning, and then hit 2nd DEL on the line with the End, then hit ENTER a lot of times.

Error Conditions

- **ERR:INVALID** occurs if this statement is used outside a program.

Related Commands

- If
- While
- Repeat
- For(

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/end>

The Eng Command

The Eng command puts the calculator in engineering notation mode. This is a variation on

Command Summary

Indicates the end of a block of statements.

Command Syntax

If *condition*
Then
statement(s)
End

While *condition*
statement(s)
End

Repeat *condition*
statement(s)
End

For(*variable,start,end[,step]*)
statement(s)
End

Menu Location

While editing a program, press:

1. PRGM to enter the PRGM menu
2. 7 to choose End, or use arrows

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

50000

scientific notation in which the exponent is restricted to be a multiple of 3 (and the mantissa can range between 1 and 1000, not including 1000 itself)

```
Eng  
Done  
12345  
12.345e3  
{1,2,3}  
{1e0 2e0 3e0}
```

Eng	50000
	Done
50000	50e3

Command Summary

Puts the calculator in engineering notation mode.

Command Syntax

Eng

Menu Location

While editing a program, press:

1. MODE to access the mode menu.
2. Use arrows and ENTER to select Eng.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

Related Commands

- [Normal](#)
- [Sci](#)
- [Float](#)
- [Fix](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/eng>

The Equ►String(Command

This command stores the contents of an equation variable (such as Y_1 or X_{1T}) to a string (one of $\text{Str}0$, $\text{Str}1$, ... $\text{Str}9$). This can be used when you want to display the equation as text (either using the Text(command on the graph screen, or the Output(or Disp commands on the home screen). For example:

```
:Equ►String(Y1,Str1  
:Text(0,0,"Y1(X)=",Str1
```

"X ² →Y ₁	Done
Equ►String(Y ₁ ,St	Done
r1	
Str1	
X ²	

Command Summary

Stores the contents of an equation variable to a string.

Command Syntax

Apart from cases in which the user has already stored to the equation variable prior to running the program, about the only situation in which you would use Equ►String(is for the output of a regression.

Advanced

You can use Equ►String((outside a program) to get the \rightarrow or " symbols in a string:

1. Type them on the home screen and press [ENTER]
2. Select 2:Quit when the **ERR:SYNTAX** comes up.
3. Press [Y=] to go to the equation editor.
4. Press [2nd] [ENTRY] to recall the symbols to Y_1
5. Now, use Equ►String(Y_1 ,Str1) to store the symbols to a string.

Related Commands

- [String►Equ\(](#)
- [expr\(](#)

See Also

- [Number to String](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/equ-string>

The ExecLib Command

Together with [OpenLib\(](#), ExecLib is used on the TI-84 Plus and TI-84 Plus SE for running routines from a Flash App library. This only works, of course, with libraries that have been specifically written for this purpose. The only such library so far is [usb8x](#), for advanced interfacing with the USB port.

Since ExecLib doesn't have any arguments, it would normally be able to run only one library routine. To get around this, usb8x uses a list passed in Ans as arguments to the command. This is most likely how any future libraries will do it as well.

The following program, which displays the version of usb8x, is an example of how to use OpenLib(and ExecLib:

```
:OpenLib(USBDRV8X
:{6
:ExecLib
:Ans(2)+.01Ans(3
```

Equ►String(*equation, string*)

Menu Location

This command is found only in the catalog. Press:

1. 2nd CATALOG to access the catalog
2. F to skip to commands starting with F
3. Scroll up to Equ►String(and select it.

Calculator Compatibility

TI-83/84/+SE

Token Size

2 bytes

```
PROGRAM:OPENLIB
:OpenLib(USBDRV8
X
:{6
:ExecLib
:Ans(2)+.01Ans(3
```

Command Summary

Calls a library routine from an application opened with [OpenLib\(](#)

Command Syntax

ExecLib

Menu Location

This command can be found in the Prgm Editor CTL menu, press:

1. Press "PRGM" while in the

Related Commands

- [OpenLib\(](#)

Program Editor.
2. Go to the last command and press "Enter".

Calculator Compatibility

TI-84+/SE

Token Size

2 bytes

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/execlib>

The expr(Command

The `expr(` command is used to evaluate an expression that's stored in a string (an expression is merely anything that returns a value - of any type). Expressions are occasionally stored to strings, rather than evaluated outright, so that their value has the capacity to change when the variables stored inside them change. The `expr(` command's result depends on the kind of expression that's in the string you pass it — it may return a number, a list, a matrix, or even another string.

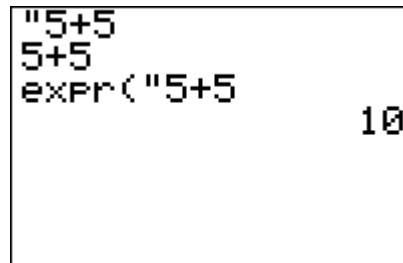
As a special case of an expression, the `expr(` command can also be used to convert a string like "123" to the number 123. Going in the reverse direction (123 to "123") is [more complicated](#).

The `expr(` command has limitations. Here are the situations in which `expr(` will not work:

- When the code in the string does not return an answer, and thus is not an expression: e.g. `expr("Line(0,0,1,1"` or `expr("prgmHELLO"` is invalid
- When the expression in the string contains an `expr(` command itself, e.g. `expr("expr(Str1"` — this will throw an [ERR:ILLEGAL NEST](#) error.
- In place of a variable (rather than an expression), e.g. `5→expr("X"` isn't a substitute for `5→X` because `expr("X"` evaluates to the value of X and not to X itself.

Advanced Uses

`expr(` is often used in conjunction with the [Input](#) command to prompt the user to enter a list. Although the `Input` command can already handle lists, it requires the user to enter the opening bracket that



A screenshot of a TI-84 calculator's display. The screen shows the following sequence of inputs and outputs:
"5+5
5+5
expr("5+5
10
The first two lines are the input of the expression "5+5" followed by the command expr("5+5". The third line is the output of the command, which is the number 10.

Command Summary

Returns the value of a string that contains an expression.

Command Syntax

`expr(string)`

Menu Location

This command is found only in the Catalog. Press:

1. 2ND CATALOG to enter the catalog
2. F to go to commands starting with F
3. Scroll up a bit to `expr(`.

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

signifies a list. With `expr()`, this can be avoided.

Instead of:

```
:Input L1
```

Use this:

```
:Input Str1  
:expr ("{"+Str1→L1
```

Optimization

Evaluating an expression inside a string is more complicated than evaluating a normal expression; you should therefore try to take as much out of an `expr()` statement as possible to speed up your code. For example:

```
:expr("sum({"+Str1  
can be  
:sum(expr("{"+Str1
```

Error Conditions

- **ERR:ILLEGAL NEST** is thrown when the string to be evaluated contains an `expr()` itself.
- **ERR:INVALID** is thrown when trying to evaluate the empty string.
- **ERR:SYNTAX** is thrown when trying to evaluate a command that doesn't return a value.

Related Commands

- [sub\(\)](#)
- [inString\(\)](#)
- [length\(\)](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/expr>

The ExpReg Command

ExpReg tries to fit an exponential curve ($y=a*b^x$) through a set of points. To use it, you must first store the points to two lists: one of the x-coordinates and one of the y-coordinates, ordered so that the Nth element of one list matches up with the Nth element of the other list. L1 and L2 are the default lists to use, and the List Editor (STAT > Edit...) is a useful window for entering the points.

The calculator does this regression by taking the

```
ExpReg  
y=a*b^x  
a=1.24030075  
b=2.426469488  
r²=.922299928  
r=.9603644766
```

Command Summary

natural log `ln(` of the y-coordinates (this isn't stored anywhere) and then doing a linear regression. The result, $\ln(y)=a*x+b$, is transformed into $y=e^b(e^a)^x$, which is an exponential curve. This algorithm shows that if any y-coordinates are negative or 0, the calculator will instantly quit with `ERR:DOMAIN`.

In its simplest form, `ExpReg` takes no arguments, and fits an exponential curve through the points in L1 and L2:

```
: {9,13,21,30,31,31,34→L1  
: {260,320,420,530,560,550,590→L2  
: LnReg
```

On the home screen, or as the last line of a program, this will display the equation of the curve: you'll be shown the format, $y=a*b^x$, and the values of a and b. It will also be stored in the `RegEQ` variable, but you won't be able to use this variable in a program - accessing it just pastes the equation wherever your cursor was. Finally, the statistical variables a, b, r, and r^2 will be set as well. These latter two variables will be displayed only if "Diagnostic Mode" is turned on (see [DiagnosticOn](#) and [DiagnosticOff](#)).

You don't have to do the regression on L1 and L2, but if you don't you'll have to enter the names of the lists after the command. For example:

```
: {9,13,21,30,31,31,34→FAT  
: {260,320,420,530,560,550,590→CALS  
: ExpReg ↵FAT,↵CALS
```

You can attach frequencies to points, for when a point occurs more than once, by supplying an additional argument - the frequency list. This list does not have to contain integer frequencies. If you add a frequency list, you must supply the names of the x-list and y-list as well, even when they're L1 and L2.

Finally, you can enter an equation variable (such as Y_1) after the command, so that the curve's equation is stored to this variable automatically. This doesn't require you to supply the names of the lists, but if you do, the equation variable must come last. You can use polar, parametric, or sequential variables as well, but since the equation will be in terms of X anyway, this doesn't make much sense.

An example of `ExpReg` with all the optional arguments:

```
: {9,13,21,30,31,31,34→FAT  
: {260,320,420,530,560,550,590→CALS  
: {2,1,1,1,2,1,1→FREQ  
: ExpReg ↵FAT,↵CALS,↵FREQ, Y1
```

Calculates the best fit exponential curve through a set of points.

Command Syntax

`ExpReg [x-list, y-list, [frequency], [equation]]`

Menu Location

Press:

1. `STAT` to access the statistics menu
2. `LEFT` to access the `CALC` submenu
3. 0 to select `ExpReg`, or use arrows

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

Related Commands

- [LnReg](#)
- [PwrReg](#)
- [SinReg](#)

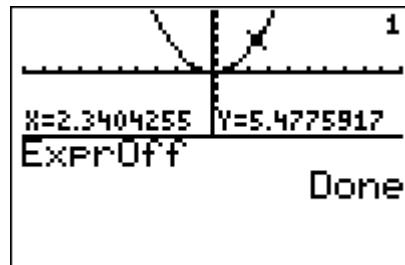
For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/expreg>

The ExprOff Command

The ExprOff command enables a "short" form of displaying the equation or plot being traced. That is, only the number of the equation or plot will be displayed, in the top right corner of the screen. When tracing a plot, the number will be prefixed with a P to distinguish it from an equation.

Related Commands

- [ExprOn](#)
- [CoordOn](#)
- [CoordOff](#)



Command Summary

The number of the equation or plot being traced is displayed in the top right corner.

Command Syntax

ExprOff

Menu Location

Press:

1. 2nd FORMAT to access the graph format menu
2. Use arrows and ENTER to select ExprOff

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/exproff>

The ExprOn Command

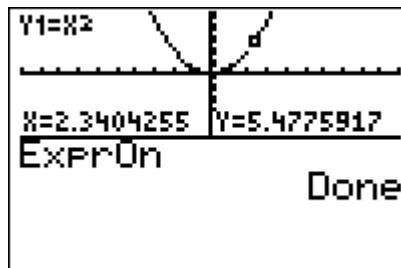
The ExprOn command enables a "long" form of displaying the equation or plot being traced.

In this mode, when tracing an equation, the equation's name and its formula are written in small font at the top of the screen. For example, when tracing Y_1 which is equal to $2X$, "Y1=2X" will be displayed.

When tracing a plot, the plot number is written, followed by the list or lists that it describes. For example, when tracing Plot1, which is a scatter plot of $\text{L}X$ and $\text{L}Y$, "P1:X,Y" will be displayed.

Related Commands

- [ExprOff](#)
- [CoordOn](#)
- [CoordOff](#)



Command Summary

The equation or plot being traced is displayed in long form at the top of the screen.

Command Syntax

ExprOn

Menu Location

Press:

1. 2nd FORMAT to access the graph format menu
2. Use arrows and ENTER to select ExprOn

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

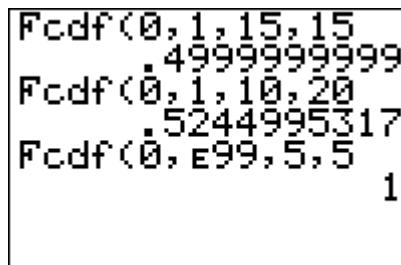
For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/expron>

The Fcdf(Command

Fcdf(is the F -distribution cumulative density function. If some random variable follows this distribution, you can use this command to find the probability that this variable will fall in the interval you supply.

The arguments *lower* and *upper* express the interval you're interested in. The arguments *numerator df* and *denominator df*, often written d_1 and d_2 , specify the F -distribution, written as $F(d_1, d_2)$.

Advanced



Command Summary

Calculates the F -distribution probability between *lower* and *upper*

Often, you want to find a "tail probability" - a special case for which the interval has no lower or no upper bound. For example, "what is the probability x is greater than 2?". The TI-83+ has no special symbol for infinity, but you can use E99 to get a very large number that will work equally well in this case (E is the decimal exponent obtained by pressing [2nd] [EE]). Use E99 for positive infinity, and -E99 for negative infinity.

Formulas

As with other continuous distributions, Fcdf(can be expressed in terms of the probability density function:

$$Fcdf(a, b, d_1, d_2) = \int_a^b Fpdf(x, d_1, d_2) dx \quad (1)$$

Related Commands

- [Fpdf\(](#)
- [ShadeF\(](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/fcdf>

for specified numerator and denominator degrees of freedom.

Command Syntax

Fcdf(lower, upper, numerator df, denominator df)

Menu Location

Press:

1. 2ND DISTR to access the distribution menu
2. 9 to select Fcdf(, or use arrows.

Press 0 instead of 9 on a TI-84+/SE with OS 2.30 or higher.

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

The Fill(Command

The Fill(command takes an existing list or matrix variable and sets all its elements to a single number. It doesn't return anything and only works on already defined variables.

```
{5}→dim(L1)
Fill(2,L1)
L1
{2 2 2 2 2}

{3,4}→dim([A])
Fill(1,[A])
[A]
[[1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]]
```

Command Summary

Fills a list or matrix with one number.

Command Syntax

Fill(value,matrix)

Menu Location

Press:

1. 2nd LIST to access the list

Optimization

When creating a new list or matrix you want to fill

with zeroes, it's better to delete it then create it with dim(, which will set all entries to 0, than to set its dimensions with dim((which may not clear what was there before) then use Fill(.

Related Commands

- dim(
- augment(

- menu.
2. RIGHT to access the OPS submenu.
 3. 4 to select Fill(, or use arrows.

Alternatively, press:

1. MATRX (83) or 2nd MATRX (83+ or higher) to access the matrix menu.
2. RIGHT to access the MATH submenu.
3. 4 to select Fill(, or use arrows.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

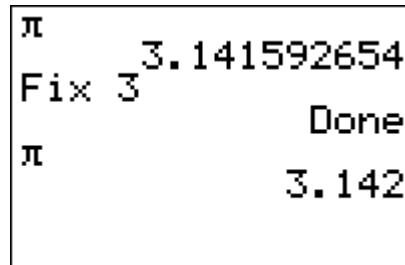
For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/fill>

The Fix Command

The Fix command puts the calculator in fixed-point display mode: all numbers will be displayed with a fixed number of digits (0-9) after the decimal, depending on the argument of Fix. This could be useful if you're trying to display potentially fractional numbers in a limited amount of space.

A note on more technical aspects: first, if more digits are available than are displayed, the calculator will round off the displayed number (but not its stored value), so 3.97 will be displayed as 4 in Fix 1 mode. Second, the Fix command can't force more than 10 significant digits to be displayed, so something like 123456789.1 will only display one decimal digit even in Fix 9 mode.

Finally, note that the Float and Fix commands only change the way numbers are displayed: they are saved in the same way in each case. Even if you're in Fix 0 mode, the calculations are not done using integers, and in general the calculations are still done using floating-point numbers no matter the number mode. The one exception is with regressions: if you store a regression to an equation in Fix N mode, it will truncate the numbers involved before storing them to the equation, and as a result, the equation will be different.



Command Summary

Puts the calculator in fixed point display mode, displaying *value* digits after the decimal.

Command Syntax

Fix *value*

Menu Location

While editing a program, press:

1. MODE to access the mode menu.
2. Use arrows to select a number 0-9 from the 2nd line.

Related Commands

- [Float](#)
- [Normal](#)
- [Sci](#)
- [Eng](#)

This will paste Fix *number*. Outside a program, it will simply put the calculator in the appropriate mode.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/fix>

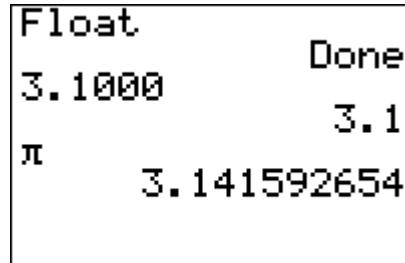
The Float Command

The Float command makes the calculator display numbers with a "floating decimal point" — only as many digits after the decimal as needed are displayed (so whole numbers, for example, are shown without any decimal points). This is the default mode, and usually the most useful.

A technicality of displaying real numbers on the calculator: A maximum of 14 significant digits are stored in a number, but only 10 of them are actually displayed (or used for comparisons) — the rest are used for additional precision. This means that if a number is displayed as a whole number, it isn't necessarily whole. For example, 1234567890.7 will be displayed as 1234567891 (rounded to 10 significant digits), and 1.0000000003 will be displayed as 1.

This makes sense from many perspectives: if you get a result of 1.0000000003 after a calculation, odds are that this should be 1, and isn't just because of a precision error. Because the extra digits are there, though, even if they're not displayed, such a number will still be invalid for functions such as Pxl-On(or sub(that want integer arguments, and this sort of error is hard to track down.

Finally, note that the Float and Fix commands only change the way numbers are displayed: they are saved in the same way in each case. Even if you're in Fix 0 mode, the calculations are not done using integers, and in general the calculations are still done using floating-point numbers no matter the number mode. The one exception is with regressions: if you store a regression to an equation in Fix N mode, it will truncate the numbers involved before storing them to the equation, and as a result, the equation will be different.



Command Summary

Puts the calculator in floating decimal display mode.

Command Syntax

Float

Menu Location

Press:

1. MODE to access the mode menu.
2. Use arrows and ENTER to select Float.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

Related Commands

- [Fix](#)
- [Normal](#)
- [Sci](#)
- [Eng](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/float>

The fMax(Command

fMax($f(var)$, var , lo , hi [, tol]) finds the value of var between lo and hi at which the maximum of $f(var)$ occurs. tol controls the accuracy of the maximum value computed. The default value of tol is 10^{-5} .

fMax(only works for real numbers and expressions. Brent's method for optimization is used for approximating the maximum value.

```
fMax(sin(X)cos(X),X,0,3)
.7853995667
```

Keep in mind that the result is the value of var , and not the value of $f(var)$. In this example, .7853995667 is not the highest possible value of $\sin(X)\cos(X)$, but rather the X-value at which $\sin(X)\cos(X)$ is the highest.

Error Conditions

- **ERR:BOUND** is thrown if the lower bound is greater than the upper bound.
- **ERR:DOMAIN** is thrown if tol is 0.
- **ERR:TOL NOT MET** is thrown if the tolerance is too small for this specific function.

Related Commands

- [fMin\(](#)
- [fnInt\(](#)
- [nDeriv\(](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/fmax>

```
fMax(5T-T^2,T,0,5
2.500001344
5Ans-Ans^2
6.25
```

Command Summary

Calculates the local maximum of a function.

Command Syntax

fMax($f(var)$, var , lo , hi [, tol])

Menu Location

While editing a program, press:

1. MATH to open the [math](#) menu
2. 7 or use arrow keys to select

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

The fMin(Command

`fMin(f(var),var,lo,hi[,tol])` finds the value of *var* between *lo* and *hi* at which the minimum of *f(var)* occurs. *tol* controls the accuracy of the minimum value computed. The default value of *tol* is 10^{-5} .

`fMin(` only works for real numbers and expressions. Brent's method for optimization is used for approximating the minimum value.

```
fMin(cos(sin(X)+Xcos(X)),X,0,2)
1.076873875
```

Keep in mind that the result is the value of *var*, and not the value of *f(var)*. In this example, 1.076873875 is not the lowest possible value of $\cos(\sin(X)+X\cos(X))$, but rather the X-value at which $\cos(\sin(X)+X\cos(X))$ is the lowest.

Advanced Uses

`fMin(` is sometimes useful in finding so-called "multiple roots" of a function. If the graph of your function appears "flat" near the root, `fMin(` might be able to find the value of the root more accurately than `solve(`.

Error Conditions

- **ERR:BOUND** is thrown if the lower bound is greater than the upper bound.
- **ERR:DOMAIN** is thrown if *tol* is 0.
- **ERR:TOL NOT MET** is thrown if the tolerance is too small for this specific function.

Related Commands

- [fMax\(](#)
- [fnInt\(](#)
- [nDeriv\(](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/fmin>

The `fnInt(` Command

`fnInt(f(var),var,a,b[,tol])` computes an approximation to the definite integral of *f* with respect to *var* from *a* to *b*. *tol* controls the accuracy of the integral computed. The default value of *tol* is 10^{-5} . `fnInt(` returns exact results for functions that are polynomials of small degree.

```
fnInt(X^2-X,X,-5,5
      .5000002687
fnInt(X^2-X,X,-5,5
      ,E-10
      .5000000192
```

Command Summary

Calculates the local minimum of a function.

Command Syntax

`fMin(f(var),var,lo,hi[,tol])`

Menu Location

While editing a program, press:

1. MATH to open the math menu
2. 6 or use arrow keys to select

Calculator Compatibility

TI-83/84/+SE

Token Size

1 byte

```
fnInt(X,X,0,10
      50
fnInt(A^3,A,-3,3
      0
```

`fnInt`(only works for real numbers and expressions. The Gauss-Kronrod method is used for approximating the integral.

Tip: Sometimes, to get an answer of acceptable accuracy out of `fnInt`(, substitution of variables and analytic manipulation may be needed.

```
fnInt(1/X,X,1,2)
      .6931471806
fnInt(ln(X),X,0,1) <a difficult exam
      -.999998347
fnInt(ln(X),X,0,1,e-11)
      -1
```

Error Conditions

- **ERR:DOMAIN** is thrown if *tol* is 0.
- **ERR:ILLEGAL NEST** is thrown if `fnInt`(occurs in the expression to be integrated.
- **ERR:TOL NOT MET** may occur if the tolerance is too small.

Related Commands

- `fMin`
- `fMax`
- `nDeriv`

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/fnint>

The FnOff Command

The `FnOff` command is used to turn off equations in the current graphing mode. When you turn off an equation, it's still defined, but isn't graphed; you can reverse this with the `FnOn` command. To turn functions on and off manually, put your cursor over the = symbol in the equation editor, and press enter.

When `FnOff` is used by itself, it will turn off all defined equations in the current graphing mode. You can also specify which equations to turn off, by writing their numbers after `FnOff`: for example, `FnOff 1` will turn off the first equation, and `FnOff 2,3,4,5` will turn off the second, third, fourth, and fifth. The numbers you give `FnOff` have to be valid equation numbers in the graphing mode. When turning equations on and off in sequence mode, use 1 for u, 2 for v, and 3 for w.

Command Summary

Approximately computes a definite integral.

Command Syntax

`fnInt(f(var),var,a,b[,tol])`

Menu Location

Press

1. Press MATH to access the math menu.
2. 9 to select `fnInt`(, or use arrows.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

FnOff 1

Command Summary

Turns off equations in the Y= editor (all of them, or only the ones specified)

Command Syntax

The most common use for FnOn and FnOff is to disable functions when running a program, so that they won't interfere with what you're doing on the graph screen, then enable them again when you're done.

Error Conditions

- ERR:DOMAIN is thrown if an equation number isn't valid in the current graphing mode, or at all.

Related Commands

- FnOn
- PlotsOn
- PlotsOff

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/fnoff>

The FnOn Command

The FnOn command is used to turn on equations in the current graphing mode. When you define an equation, it's turned on by default, but the FnOff command can turn an equation off (in which case, it's still defined, but isn't graphed). To turn functions on and off manually, put your cursor over the = symbol in the equation editor, and press enter.

When FnOn is used by itself, it will turn on all defined equations in the current graphing mode. You can also specify which equations to turn on, by writing their numbers after FnOn: for example, FnOn 1 will turn off the first equation, and FnOn 2,3,4,5 will turn the second, third, fourth, and fifth. The numbers you give FnOn have to be valid equation numbers in the graphing mode. When turning equations on and off in sequence mode, use 1 for u, 2 for v, and 3 for w.

The most common use for FnOn and FnOff is to disable functions when running a program, so that they won't interfere with what you're doing on the graph screen, then enable them again when you're done.

Error Conditions

FnOff numbers//

Menu Location

Press:

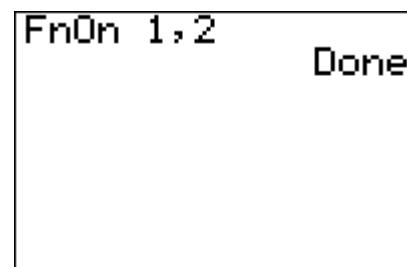
1. VARS to access the variables menu.
2. RIGHT to access the Y-VARS submenu.
3. 4 to select On/Off..., or use arrows and ENTER.
4. 2 to select FnOff, or use arrows and enter.

Calculator Compatibility

TI-83/84/+SE

Token Size

1 byte



Command Summary

Turns on equations in the Y= editor (all of them, or only the ones specified)

Command Syntax

FnOn numbers//

Menu Location

Press:

1. VARS to access the variables menu.
2. RIGHT to access the Y-VARS submenu.

- **ERR:DOMAIN** is thrown if an equation number isn't valid in the current graphing mode, or at all.

Related Commands

- FnOff
- PlotsOn
- PlotsOff

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/fnon>

The For(Command

A For(loop is generally used to do something a specific number of times, or to go through each one of a bunch of things (such as elements of a list, or the pixels of your screen). Of all the loops, it's the most complicated. Its syntax:

```
For(variable,start,end[,step]
statement(s)
End
```

What the loop does:

1. Set *variable* to equal *start*.
2. Run all the *statement(s)*.
3. Increase the value of *variable* by *step* (or by 1, if you didn't provide a *step*)
4. As long as *variable* is no greater than *end* (or no less than, if *step* is negative), go back to step 2.

In other words: a **For(loop repeats its contents once for every value of variable between start and end.**

This is perhaps best explained with an example. The following code will display the numbers 1 to 10, in order:

```
:For(A,1,10)
:Disp A
:End
```

Now, all of this could be done with a Repeat or While command and some manipulation, except that

3. 4 to select On/Off..., or use arrows and ENTER.
4. ENTER to select FnOn.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

PROGRAM: EXAMPLE
:For(A,1,9,2
:Disp A
:End

Command Summary

Executes some commands many times, with a variable increasing from *start* to *end* by *step*, with the default value *step*=1.

Command Syntax

```
For(variable,start,end[,step])
statement(s)
End
```

Menu Location

While editing a program press:

1. PRGM to enter the PRGM menu
2. 4 to choose For(, or use arrows
3. 7 to choose End, or use arrows

Calculator Compatibility

TI-83/84/+/SE

Token Size

this is faster because it's a single command. Still, why have a separate command for something that seems so specific and arbitrary? Well, it's because For(has so many uses!

1 byte

- Do something to each element of a list, matrix, or string.
- Draw several similar objects on the graph screen.
- Create animations.
- Easily add the possibility of levels to many games.
- Any number of other things...

An advanced note: each time the program enters a For(loop, the calculator uses 43 bytes of memory to keep track of this. This memory is given back to you as soon as the program reaches End. This isn't really a problem unless you're low on RAM, or have a lot of nested For statements. However, if you use Goto to jump out of a For(loop, you lose those bytes for as long as the program is running — and if you keep doing this, you might easily run out of memory, resulting in ERR:MEMORY.

Advanced Uses

Sometimes you want to exit out of a For(loop when it hasn't finished. You can do this by storing something at least equal to the *end* value to the variable you used in the For(loop. For example:

```
:For(A,1,100)
<some code>
:If <condition for exiting out>
:100→A
:End
```

For(can also be used to create a delay in the program for some time:

```
:For(A,1,200)
:End
```

Change the 200 to a higher or lower number to create a longer or shorter delay.

Actually the rand command can be used to achieve a delay, but at a smaller size. However, only the For(command can be used if you want to do something, such as a simple animation, during the delay.

You can combine two or more For(loops to run a block of statements once for every permutation of several variables. For example:

```
:For(A,1,50)
:For(B,1,50)
:(some code)
:End
:End
```

This will run (some code) 2500 times — once for every combination of a value of A from 1 to 50 and a value of B from 1 to 50.

There's a standard way to exclude repetitions if the order of the variables doesn't matter (for example, if A=30, B=40 is the same situation as A=40, B=30 in the example above). In this case, the beginning of the loop should be changed to:

```
:For(A,1,50)  
:For(B,1,A)
```

Optimization

The seq(command, or simple math, can often be used in place of the For(command when dealing with lists. For example:

```
:For(A,1,dim(L1  
:A→L1(A  
:End  
can be  
:seq(A,A,1,dim(L1→L1
```

and

```
:For(A,1,dim(L1  
:1+L1(A→L1(A  
:End  
can be  
:1+L1→L1
```

One rather strange optimization when using For(loops is actually leaving on the ending parenthesis of the For(loop in certain cases. If you don't do this, the following cases will be processed **much** slower inside the loop:

- IS>(and DS<((no matter if the following command is skipped or not).
- A lone If without an accompanying Then, but **only** when the condition is false (If with a true condition is unchanged).

If the condition of the If command can be either true or false (as in most actual cases), you should still add a closing parenthesis because the difference is so great.

This difference is responsible for the supposed difference between speed of If commands with or without an accompanying Then. There is actually no difference when you account for this factor — the commands are the same speed if you close the parenthesis.

An example use of this optimization:

```
:For(I,1,1200  
:If not(fPart(I/5):Disp I  
:End
```

should instead be

```
:For(I,1,1200)
:If not(fPart(I/5):Disp I
:End
```

Command Timings

Using a For(loop when it fits your purpose is much faster than adapting a While or Repeat loop to do so. Conclusion: For(loops are good!

Error Conditions

- **ERR:INCREMENT** is thrown if the increment of the For(loop is 0.
- **ERR:INVALID** occurs if this statement is used outside a program.
- **ERR:UNDEFINED** is thrown if you DelVar the loop variable while inside the loop.

Related Commands

- [Repeat](#)
- [While](#)
- [If](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/for>

The fPart(Command

fPart(*value*) returns the fractional part of *value*. Also works on complex numbers, lists and matrices.

```
fPart(5.32)           .32
fPart(4/5)            .8
fPart(-5.32)          -.32
fPart(-4/5)           -.8
```

```
fPart(π)              1415926536
fPart(-π)             -1415926536
9fPart(12121/9)       7
```

Advanced Uses

fPart(), along with [int\(\)](#) or [iPart\(\)](#), can be used for integer [compression](#).

Also, fPart() is an easy way to find A mod B (the positive remainder when A is divided by B).

Command Summary

Returns the fractional part of a value.

Command Syntax

fPart(*value*)

Menu Location

Press:

1. MATH to access the [math](#)

```
:B(A<0)+iPart(BfPart(A/B))
```

If A is guaranteed to be positive, the following shorter code can be used, omitting B(A<0):

```
:iPart(BfPart(A/B))
```

Finally, the easiest way to check if a number is a whole number is not(fPart(X):

```
:If not(fPart(X:Then  
: // X is an integer  
:Else  
: // X is not an integer  
:End
```

You can use this, for example, to check if a number is divisible by another: if X is divisible by N, then X/N is a whole number. This is useful when you want to find the factors of a number.

Warning: when storing values with repeating decimals and later multiplying them to see if a number makes it an integer it can return a value of 1 or -1 instead of 0 even if it is an integer.

Example: if you store 1/3 as X and then do fpart(3x) it will return 1 instead of 0. This is because fpart(.999...) results in .999... and then rounds to 1 when displaying rather than rounding to 1.0 and then displaying the fpart(as 0.

Optimization

Often you want to find the value of a-1 mod b — this occurs, for example, in movement routines with wraparound. However, the problem is that if a=0, a-1 will be negative. Rather than use the longer version of the modulo routine, you might replace subtracting 1 with adding (b-1). This will have the same result, but without sign problems.

Related Commands

- [int\(](#)
- [iPart\(](#)
- [round\(](#)

See Also

- [Compression](#)
- [Number Factorization](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/fpart>

menu.

2. RIGHT to access the NUM submenu.
3. 4 to select fPart(, or use arrows.

Calculator Compatibility

TI-83/84/+SE

Token Size

1 byte

The Fpdf(Command

`Fpdf`(is the F-distribution probability density function.

Since the F-distribution is continuous, the value of `Fpdf`(doesn't represent an actual probability - in fact, one of the only uses for this command is to draw a graph of the distribution. You could also use it for various calculus purposes, such as finding inflection points.

The command takes 3 arguments: x is the point at which to evaluate the function (when graphing, use X for this argument), *numerator df* and *denominator df* are the numerator degrees of freedom and denominator degrees of freedom respectively (these specify a single `Fpdf`(curve out of an infinite family).

The F-distribution is used mainly in significance tests of variance.

Formulas

The value of the `Fpdf`(is given by

(1)

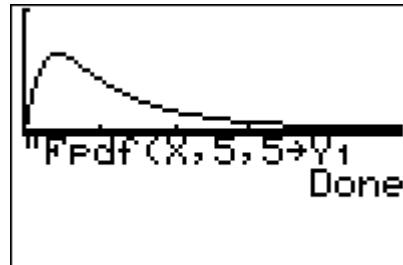
$$\text{Fpdf}(x, d_1, d_2) = \frac{\left(\frac{d_1 x}{d_1 x + d_2}\right)^{d_1/2} \left(1 - \frac{d_1 x}{d_1 x + d_2}\right)^{d_2/2}}{x \text{B}(d_1/2, d_2/2)}$$

Here, B is the Beta function.

Related Commands

- `Fcdf`(
- `ShadeF`(

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/fpdf>



Command Summary

Evaluates the F-distribution probability density function at a point.

Command Syntax

`Fpdf(x , numerator df, denominator df)`

Menu Location

Press:

1. 2ND DISTR to access the distribution menu
2. 8 to select `Fpdf`(, or use arrows.

Press 9 instead of 8 on a TI-84+/SE with OS 2.30 or higher.

Calculator Compatibility

TI-83/84+/SE

Token Size

2 bytes

The ►Frac Command

►Frac attempts to display the input in fraction form. It only works on the home screen outside a program, or with the Disp and Pause commands in a program. It takes at least 12 decimal places of a non-terminating decimal to find the corresponding fraction. The decimal input is returned if ►Frac fails to find the fraction form.

For a better algorithm for finding fractions, see the Decimal to Fraction routine.

```
.333►Frac  
.333  
.333333333333►Frac  
1/3
```

Related Commands

- ►Dec

See Also

- Decimal to Fraction

7/2	3.5
7/2►Frac	7/2
7/2►Dec	3.5

Command Summary

Displays the fractional value of a number

Command Syntax

Decimal►Frac

Menu Location

While editing a program, press:

1. MATH to open the math menu
2. ENTER or 1 to select.

Calculator Compatibility

TI-83/84/+SE

Token Size

1 byte

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/frac>

The Full Command

The Full command cancels the effects of either Horiz or G-T.

Full is usually used either at the beginning and/or ending of a program. It is used at the beginning to ensure that the screen mode is Full, the standard setting. It is used at the end if the screen mode was changed in the middle of the program.

```
:Full
```



Command Summary

Sets the screen mode to FULL.

Related Commands

- [G-T](#)
- [Horiz](#)

Command Syntax

Full

Menu Location

In the BASIC editor,

1. Press [MODE]
2. Press [DOWN] seven times
3. Press [ENTER] to insert Full

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/full>

The Func Command

The Func command enables function graphing mode. This is usually unnecessary in a program, but if you want to graph a Y# equation, you'd want to make sure the calculator is in function mode first.

In function mode, you can graph equations where y (the vertical coordinate) is a function of x (the horizontal coordinate). This mode is most commonly discussed in algebra and single-variable calculus courses. Many curves, such as a parabola, have simple expressions when written in the form $y=f(x)$.

However, in function mode, many expressions cannot be graphed at all. For example, a circle can't be easily graphed in function mode, since for some x-values, there are two y-values. Using two functions, you can achieve a circle, but it will still require a friendly graphing window to display perfectly.

Many calculator features are specifically targeted at function mode graphing. For example, two graphing styles (see GraphStyle()) can be only used with function mode. The DrawF command draws a function in graphing mode.

Advanced Uses

```
Plot1 Plot2 Plot3  
Y1=X2  
Y2=  
Y3=  
Y4=  
Y5=  
Y6=  
Y7=
```

Command Summary

Enables function graphing mode.

Command Syntax

Func

Menu Location

While editing a program, press:

1. MODE to access the mode menu.
2. Use arrows to select Func.

Calculator Compatibility

TI-83/84/+/SE

The window variables that apply to function mode are:

- **Xmin** — Determines the minimum X-value shown on the screen.
- **Xmax** — Determines the maximum X-value shown on the screen.
- **Xscl** — Determines the horizontal space between marks on the X-axis in AxesOn mode or dots in GridOn mode.
- **Ymin** — Determines the minimum Y-value shown on the screen.
- **Ymax** — Determines the maximum Y-value shown on the screen.
- **Yscl** — Determines the vertical space between marks on the Y-axis in AxesOn mode or dots in GridOn mode.
- **Xres** — Determines the pixel distance between points used for graphing. This is a value 1-8: 1 for best quality, 8 for best speed.

Token Size

1 byte

Related Commands

- Param
- Polar
- Sq

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/func>

The GarbageCollect Command

A bit of a preamble: unlike RAM, which is the easy-to-access memory, Flash ROM (the archive), used for long-term storage on the 83+ and higher, can't be written to easily. Skipping over technicalities, what's written in the archive once is semi-permanent, and can't be written to again unless an entire 64KB sector of memory is erased.

As a result, when you delete a variable from archive, the calculator doesn't delete it immediately (there may be other, good variables in the same block that would get erased as well), it just marks it as deleted. Similarly, when you unarchive a variable, its data is copied to RAM and the original is marked as deleted.

Naturally, this can't be done forever: sooner or later you'll run out of space in the archive because all of it is taken up by these "garbage variables". At this point, the calculator does something known as "garbage collecting". It copies the actually-used variables in each sector to a backup sector (set aside just for this purpose), then erases it; the process is repeated for the other sectors.

Additionally, the variables are rearranged so that they aren't spread out all over the place; this makes it more likely that a spot will be found for large variables.



Command Summary

Clears up 'garbage' that comes from unarchiving or deleting archived files.

Command Syntax

GarbageCollect

Menu Location

This command can only be found in the catalog. Press:

1. 2nd CATALOG to access the command catalog
2. G to skip to commands starting with G
3. ENTER to select

While "garbage collecting" will be done automatically when it's absolutely necessary, this may be a time-consuming process at that stage. Instead, you can call the GarbageCollect command yourself periodically (how often depends on your calculator habits, but generally once a month or so could work) to keep the Flash ROM in a semi-neat state, and then it will be a fairly quick process.

During garbage collection, a menu will appear that asks you "Garbage Collect?", giving you the options No and Yes. If you didn't select the GarbageCollect command yourself, it's highly recommended to select Yes. If you did select it, you probably want to garbage collect, so you should also select Yes. At that point, the message "Garbage collecting..." will be displayed for some time, and then the process will end.

Advanced Uses

To avoid garbage collecting often, reduce the amount of times you archive and unarchive variables. There's also the consideration that too many writes to the Flash ROM (which are directly related to the number of GarbageCollects you do) can, in theory, wear it out. This probably would take much longer than anyone's used a TI-83+ calculator so far, though, and in all probability you don't really have to worry about this.

Related Commands

- [Archive](#)
- [UnArchive](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/garbagecollect>

The gcd(Command

Returns the greatest common divisor (GCD) of two nonnegative integers. Also works on lists.

```
gcd(8,6)  
2  
gcd({9,12},6)  
{3 6}  
gcd({14,12},{6,8})  
{2 4}
```

gcd(12,15)	3
gcd(0,25)	25
gcd(12,gcd(18,36))	6

Error Conditions

- **ERR:DIM MISMATCH** is thrown if the arguments are two lists that don't have the same number of elements.
- **ERR:DOMAIN** is thrown if the arguments aren't positive integers (or lists of positive

GarbageCollect

Calculator Compatibility

TI-83+/84+/SE
(not available on the regular TI-83)

Token Size

2 bytes

Command Summary

Finds the greatest common divisor of two values.

Command Syntax

gcd(value1, value2)

Menu Location

integers) less than 1e12.

Related Commands

- [lcm\(](#)

Press:

1. MATH to access the math menu.
2. RIGHT to access the NUM submenu.
3. 9 to select gcd(, or use arrows.

Calculator Compatibility

TI-83/84+/SE

Token Size

2 bytes

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/gcd>

The `geometcdf(` Command

This command is used to calculate cumulative geometric probability. In plainer language, it solves a specific type of often-encountered probability problem, that occurs under the following conditions:

1. A specific event has only two outcomes, which we will call "success" and "failure"
2. The event is going to keep happening until a success occurs
3. Success or failure is determined randomly with the same probability of success each time the event occurs
4. We're interested in the probability that it takes **at most** a specific amount of trials to get a success.

For example, consider a basketball player that always makes a shot with 1/4 probability. He will keep throwing the ball until he makes a shot. What is the probability that it takes him no more than 4 shots?

1. The event here is throwing the ball. A "success", obviously, is making the shot, and a "failure" is missing.
2. The event is going to happen until he makes the shot: a success.
3. The probability of a success - making a shot - is 1/4
4. We're interested in the probability that it takes at most 4 trials to get a success

```
geometcdf(.5,1      .5  
geometcdf(.5,2      .75  
geometcdf(.5,30     .999999991
```

Command Summary

Calculates the cumulative geometric probability for a single value

Command Syntax

`geometcdf(probability, trials)`

Menu Location

Press:

1. 2ND DISTR to access the distribution menu
2. ALPHA E to select `geometcdf(`, or use arrows.

Press ALPHA F instead of ALPHA E on a TI-84+/SE with OS 2.30 or higher.

The syntax here is `geometcdf(probability, trials)`. In this case:

```
: geometcdf(1/4, 4
```

This will give about .684 when you run it, so there's a .684 probability that he'll make a shot within 4 throws.

Note the relationship between `geometpdf()` and `geometcdf()`. Since `geometpdf()` is the probability it will take **exactly** N trials, we can write that $\text{geometcdf}(1/4, 4) = \text{geometpdf}(1/4, 1) + \text{geometpdf}(1/4, 2) + \text{geometpdf}(1/4, 3) + \text{geometpdf}(1/4, 4)$.

Formulas

Going off of the relationship between `geometpdf()` and `geometcdf()`, we can write a formula for `geometcdf()` in terms of `geometpdf()`:

$$\text{geometcdf}(p, n) = \sum_{i=1}^n \text{geometpdf}(p, i) = \sum_{i=1}^n p(1-p)^{n-1} \quad (1)$$

(If you're unfamiliar with sigma notation, $\sum_{i=1}^n$ just means "add up the following for all values of i from 1 to n")

However, we can take a shortcut to arrive at a much simpler expression for `geometcdf()`. Consider the opposite probability to the one we're interested in, the probability that it will **not** take "at most N trials", that is, the probability that it will take more than N trials. This means that the first N trials are failures. So $\text{geometcdf}(P, N) = (1 - \text{"probability that the first N trials are failures"})$, or:

$$\text{geometcdf}(p, n) = 1 - (1 - p)^n \quad (2)$$

Related Commands

- [binompdf\(\)](#)
- [binomcdf\(\)](#)
- [geometpdf\(\)](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/geometcdf>

The `geometpdf()` Command

This command is used to calculate geometric probability. In plainer language, it solves a specific type of often-encountered probability problem, that occurs under the following conditions:

1. A specific event has only two outcomes, which

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

```
geometpdf(.5, 1      .5
geometpdf(.5, 2      .25
geometpdf(.5, 30     .025
```

we will call "success" and "failure"

2. The event is going to keep happening until a success occurs
3. Success or failure is determined randomly with the same probability of success each time the event occurs
4. We're interested in the probability that it takes a specific amount of trials to get a success.

For example, consider a basketball player that always makes a shot with $\frac{1}{3}$ probability. He will keep throwing the ball until he makes a shot. What is the probability that it takes him 3 shots?

1. The event here is throwing the ball. A "success", obviously, is making the shot, and a "failure" is missing.
2. The event is going to happen until he makes the shot: a success.
3. The probability of a success - making a shot - is $\frac{1}{3}$
4. We're interested in the probability that it takes 3 trials to get a success

The syntax here is `geometpdf(probability, trials)`. In this case:

```
:geometpdf(1/3,3)
```

This will give about .148 when you run it, so there's a .148 probability that it will take him 3 shots until he makes one (he'll make it on the 3rd try).

Formulas

The value of `geometpdf()` is given by the formula

$$\text{geometpdf}(p, n) = p(1 - p)^{n-1} \quad (1)$$

This formula can be intuitively understood: the probability that the first success is the n th trial is the probability of getting a success - p - times the probability of missing it the first $n-1$ times - $(1 - p)^{n-1}$.

For the trivial value of $n=0$, however, the above formula gives the incorrect value of 1. It should actually be 0, since the first success can never be the 0th trial. However, since you're not likely to ever be interested in this probability, this drawback doesn't really matter.

Related Commands

- [binompdf\(\)](#)
- [binomcdf\(\)](#)
- [geometcdf\(\)](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/geometpdf>

9.31322070E-10

Command Summary

Calculates the geometric probability for a single value

Command Syntax

`geometpdf(probability, trials)`

Menu Location

Press:

1. 2ND DISTR to access the distribution menu
2. ALPHA D to select `geometpdf()`, or use arrows.

Press ALPHA E instead of ALPHA D on a TI-84+/SE with OS 2.30 or higher.

Calculator Compatibility

TI-83/84+/SE

Token Size

2 bytes

The GetCalc(Command

The GetCalc(command allows you to make multiplayer games, where two calculators communicate with each other across a link cable that is connected between them. The GetCalc(command can only receive one variable from another calculator, and the variable can be any variable (a real, list, matrix, string, etc.). The calculator doesn't exchange variable values when the variable is received, but instead replace the variable of the same name on the receiving calculator.

For the GetCalc(command to work correctly, the sending calculator must be in a power-saving state and it cannot be executing an assembly program. (The sending calculator is the one which is *not* executing the GetCalc(command.) The two main commands that you should use to ensure this are Pause and Menu(; however, any command that is waiting for user input will also work perfectly fine (such as Prompt and Input).

The GetCalc(command behaves a little differently in the older TI-83 models. If the sending calculator is idle with the Pause or Menu(command, it will automatically "press enter" when the receiving calculator executes GetCalc(. This can be frustrating when in a menu, because it prevents the user's opportunity to make a selection.

However, this can make real-time gaming more possible if used in conjunction with the Pause command. When the receiving calculator receives the variable, it could then execute the Pause command, while the sending calculator automatically exits the power-saving state and could then perform the GetCalc(command. All models after the TI-83 do not automatically exit their power-saving states.

Advanced Uses

The TI-84+ and TI-84+SE will use the USB port if it is connected to a USB cable, otherwise they will use the I/O port. However, you can specify which port you want to use by putting a number after the variable as GetCalc's second argument: zero to use the USB port if connected to a USB cable, one to use the USB port without checking to see if it's connected, and three to use the I/O port.

Related Commands

GetCalc(X	Done
GetCalc(L1	Done

Command Summary

Gets a variable from another calculator.

Command Syntax

GetCalc(*variable*)

(84+ and 84+SE only)
GetCalc(*variable*,*portflag*)

Menu Location

While editing a program, press:

1. PRGM to enter the PRGM menu
2. RIGHT to enter the I/O menu
3. 9 to choose GetCalc(, or use arrows

Calculator Compatibility

TI-83/84/+SE

Token Size

2 bytes

- [Get\(](#)
- [Send\(](#)

See Also

- [Multiplayer](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/getcalc>

The Get(Command

The Get(command is meant for use with the CBL (Calculator Based Laboratory) device, or other compatible devices. When the calculator is connected by a link cable to such a device, Get(*variable*) will read data from the device and store it to *variable*. Usually, this data is a list, and so you want to Get(L1) or some other list variable.

Advanced Uses

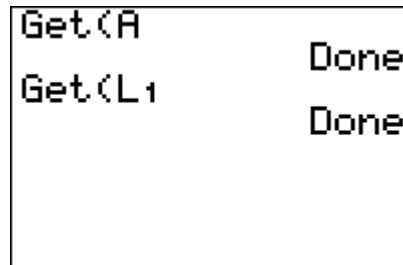
In fact, the Get(command can also be used for linking two calculators, in which case it functions precisely like [GetCalc\(\)](#). This is probably for compatibility with the TI-82, which used Get(rather than GetCalc(for linking two calculators. However, since this isn't a documented feature (in fact, your TI-83+ manual will insist that Get(**cannot** be used in this way), it isn't guaranteed to work with future calculator versions.

Optimization

Nevertheless, using Get(instead of GetCalc(will make your program smaller, and probably preserve functionality.

Related Commands

- [GetCalc\(](#)
- [Send\(](#)



Command Summary

Gets a variable's value from a connected calculator or CBL device.

Command Syntax

`Get(variable)`

Menu Location

While editing a program, press:

1. PRGM to access the program menu.
2. RIGHT to access the I/O menu.
3. ALPHA A to select Get(.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/get>

The getDate Command

The getDate command returns the current date that the clock has on the TI-84+/SE calculators in list format — {year, month, day}. You can store this list to a variable for later use, or manipulate it the same way you do with other lists. Of course, this command only works if the date has actually been set, so you should use the setDate(command before using it.

An interesting note about this command is that you cannot use the standard listname(var) to access elements - If you try, it multiplies each element of the clock by the number.

```
getDate  
    (2008 11 14)  
getDate(2  
    (4016 22 28)
```

■

Related Commands

- getDtFmt
- getDtStr(
- setDate(
- setDtFmt(

```
PROGRAM: SHOWDATE  
:ClrList L1  
:getDate→L1  
:Disp "YEAR:",L1(1), "MONTH:",L1(2), "DAY:",L1(3)
```

Command Summary

Returns a list with the current date that the clock has on the TI-84+/SE.

Command Syntax

getDate→Variable

Menu Location

This command can only be found in the catalog. Press:

1. 2nd CATALOG to enter the command catalog
2. g to skip to commands starting with G
3. Scroll down to getDate and select it

Calculator Compatibility

TI-84+/SE

Token Size

2 bytes

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/getdate>

The getDtFmt Command

The getDtFmt(command returns the current date format of the clock on the TI-84+/SE calculators as an integer. There are three different date formats available: 1 (M/D/Y), 2 (D/M/Y), and 3 (Y/M/D). You can store this value to a variable for later use. Of course, this command only works if the date format

```
PROGRAM: DATEFMT  
:getDtFmt→N  
:If N=1  
:Disp "M/D/Y"  
:If N=2  
:Disp "D/M/Y"
```

has actually been set, so you should use the [setDtFmt\(\)](#) command before using it.

Related Commands

- [getDate](#)
- [setDate\(](#)
- [setDtFmt\(](#)
- [getDtStr\(](#)

```
:Disp "D/M/Y"
:If N=3
:Disp "Y/M/D"
```

Command Summary

Returns the date format of the clock on the TI-84+/SE.

Command Syntax

`getDtFmt→Variable`

Menu Location

This command can only be found in the catalog. Press:

1. 2nd CATALOG to enter the command catalog
2. g to skip to commands starting with G
3. Scroll down to `getDtFmt(` and select it

Calculator Compatibility

TI-84+/SE

Token Size

2 bytes

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/getdtfmt>

The `getDtStr(` Command

The `getDtStr(` command returns the current date of the clock on the TI-84+/SE calculators as a string based on the date format that is specified. There are three different date formats available: 1 (M/D/Y), 2 (D/M/Y), or 3 (Y/M/D). You can store this value to a string variable for later use, or manipulate it the same way you do with other strings. Of course, this command only works if the date format has actually been set, so you should use the [setDtFmt\(\)](#) command before using it.

Related Commands

- [getDate](#)
- [getDtFmt](#)
- [setDate\(](#)
- [setDtFmt\(](#)

```
PROGRAM:DATESTR
:getDtFmt→N
:Disp getDtStr(N
)
```

Command Summary

Returns the current date of the clock on the TI-84+/SE as a string.

Command Syntax

`getDtStr(value)→variable`

Menu Location

This command can only be found in the catalog. Press:

1. 2nd CATALOG to enter the command catalog
2. g to skip to commands starting with G
3. Scroll down to getDtStr(and select it

Calculator Compatibility

TI-84+/SE

Token Size

2 bytes

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/getdtstr>

The getKey Command

The getKey command returns the value of the last key pressed since the last time getKey was executed. Reading keypresses with getKey allows a program to transfer control to the user, and you can combine getKey with other commands to create menus, movement, or whatever else you want.

Every key has a number assigned to it, except for ON (which is used for breaking out of programs). The numbering system consists of a row and column: the rows go from one to ten, starting at the top; and the columns go from one to six, starting from the left. You just put the row and column together to get the key's number — for example, the ENTER key is located in row 10, column 5, therefore its value is 105. The arrow keys look like they would numbered separately from the other keys, but they actually follow this pattern as well. See the key codes page for a picture of the key codes on the calculator.

The value of getKey is cleared every time you read from it, until a new key is pressed. For this reason, except in very rare cases, you do not want to use the value of getKey in an expression directly, but store it to a variable first. It is also common to use getKey inside of a loop, so that the program can wait for the user to press a key.

```
PROGRAM: EXAMPLE
:ClrHome
:Disp "PRESS CLE
AR
:Repeat K=45
:getKey→K
:End■
```

Command Summary

Returns the numerical code of the last key pressed, or 0 if no key is pressed.

Command Syntax

getKey[→*Variable*]

Menu Location

While editing a program, press:

1. PRGM to enter the PRGM menu
2. RIGHT to enter the I/O menu
3. 7 to choose getKey or use arrows

Calculator Compatibility

```
:Repeat Ans  
:getKey→K  
:End
```

TI-83/84/+SE

Token Size

1 byte

Advanced Uses

You can put getKey in the condition of a loop, to make the loop repeat until any key or a particular key is pressed by the user. The same thing can be done with conditionals as well. This is useful if you don't want to store getKey to a variable, but you still want to have the user press a key. This works because of the 'true' or 'false' way that TI-Basic is interpreted.

```
:Repeat max(getKey={24,25,26,34}  
:End
```

Unlike the other keys, the arrow and DEL keys can actually be held down, which will cause the key to keep being repeated until it is unpressed. This functionality is very useful in games where the user needs to repeatedly press a key to move or shoot, although it does completely disable the other keys from being able to be pressed (which is important in multiplayer games, where everybody must share the keys).

Sometimes your program may do something for several seconds without user input (say, playing an animation), then pause and wait for a key to be pressed. The problem is that if a key is pressed during the animation, the next getKey will return the value of that key, and any loop set up to wait for a key press will exit immediately. The solution is to run a "dummy" getKey just before the loop begins — its value won't be used for anything, and it will reset the value of getKey to 0. This can also be used to clear keypresses meant for loading programs from inside a shell.

Error Conditions

- ERR:INVALID occurs if this statement is used outside a program.

Related Commands

- Input
- Prompt

See Also

- Key Codes
- Custom Menus
- Movement in Maps

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/getkey>

The getKey() Command

The getKey() command returns the key code of the last keypress. If no key was pressed since the program, function, or expression started running, or since the last getKey() command, getKey() returns 0.

The keypresses that getKey() deals with factor in modifier keys, such as 2nd or alpha. Because of this, it will not respond to the modifier keys pressed by themselves.

This example code using getKey() is commonly used in programs that wait for the user to press a key:

```
:0→key
:While key=0
:getKey()→key
:EndWhile
```

Advanced Uses

Although the key codes are given in a table on this website, and are listed in your manual, it may be more convenient to write a short function to return key codes for you:

```
:keycode()
:Func
:Local k
:0→k
:While k=0
:getKey()→k
:EndWhile
:k
:EndFunc
```

If you run the function keycode(), it will wait for you to press a key. When you press it, it will return the key code. This function may also be a convenient subroutine in a program that requires waiting for a key in several different places.

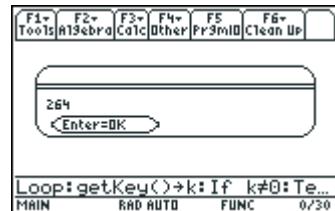
Related Commands

- [Input](#)
- [InputStr](#)
- [Prompt](#)

See Also

- [Key Codes](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/68k:getkey>



Command Summary

Returns the last keypress.

Command Syntax

getKey()

Menu Location

From the program editor toolbar:

1. Choose F3 - I/O
2. Press 7 to paste getKey()

Calculator Compatibility

This command works on all calculators.

Token Size

3 bytes total:

- 0xE3 ('extra command' tag)
- 0x02 (command identifier)
- 0xE5 ('end of arguments' tag)

The getTime Command

The `getTime` command returns the current time that the clock has on the TI-84+/SE calculators in list format — `{hour, minute, second}`. You can store this list to a variable for later use, or manipulate it the same way you do with other lists. Of course, this command only works if the time has actually been set, so you should use the `setTime(` command before using it.

An interesting note about this command is that you cannot use the standard listname(var) to access elements - If you try, it multiplies each element of the clock by the number.

```
getTime {22 31 28}
getTime(2 {44 62 62}
■
```

Related Commands

- [getTmFmt](#)
- [getTmStr\(](#)
- [setTime\(](#)
- [setTmFmt\(](#)

```
PROGRAM:TIME
:ClrList L1
:getTime→L1
:Disp "HOUR:",L1
(1),"MINUTE:",L1
(2),"SECOND:",L1
(3)
```

Command Summary

Returns a list with the current time that the clock has on the TI-84+/SE.

Command Syntax

`getTime→Variable`

Menu Location

This command can only be found in the catalog. Press:

1. 2nd CATALOG to enter the command catalog
2. g to skip to commands starting with G
3. Scroll down to `getTime` and select it

Calculator Compatibility

TI-84+/SE

Token Size

2 bytes

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/gettime>

The getTmFmt Command

The `getTmFmt(` command returns the current time

format of the clock on the TI-84+/SE calculators as an integer. There are two different time formats available: 12 (12 hour) and 24 (24 hours). You can store this value to a variable for later use. Of course, this command only works if the time format has actually been set, so you shold use the setTmFmt(command before using it.

Related Commands

- getTime
- setTime(
- setTmFmt(
- getTmStr(

```
PROGRAM:TIME
:getTmFmt→L
:If L=12
:Disp "AM/PM"
:If L=24
:Disp "MILITARY"
```

Command Summary

Returns the time format of the clock on the TI-84+/SE.

Command Syntax

`getTmFmt→Variable`

Menu Location

This command can only be found in the catalog. Press:

1. 2nd CATALOG to enter the command catalog
2. g to skip to commands starting with G
3. Scroll down to `getTmFmt(` and select it

Calculator Compatibility

TI-84+/SE

Token Size

2 bytes

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/gettmfmt>

The `getTmStr(` Command

The `getTmStr(` command returns the current time of the clock on the TI-84+/SE calculators as a string based on the time format that is specified. There are two different time formats available: 12 (12 hour) or 24 (24 hour). You can store this value to a string variable for later use, or manipulate it the same way you do with other strings. Of course, this command only works if the time format has actually been set, so you should use the setTmFmt(command before using it.

Related Commands

```
PROGRAM:TIME
:getTmFmt→N
:Disp getTmStr(N
)
```

Command Summary

Returns the current time of the

- [getTime](#)
- [getTmFmt](#)
- [setTime\(](#)
- [setTmFmt\(](#)

clock on the TI-84+/SE as a string.

Command Syntax

`getTmStr(value)→variable`

Menu Location

This command can only be found in the catalog. Press:

1. 2nd CATALOG to enter the command catalog
2. g to skip to commands starting with G
3. Scroll down to `getTmStr(` and select it

Calculator Compatibility

TI-84+/SE

Token Size

2 bytes

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/gettmstr>

The Goto Command

The Goto command is used together with the Lbl command to jump (or branch) to another place in a program. When the calculator executes a Goto command, it stores the label name in memory, and then searches from the beginning of the program for the Lbl command with the supplied name. If it finds it, it continues running the program from that point; otherwise, if the label does not exist, it throws an [ERR: LABEL](#) error.

Label names can be either one or two characters long, and the only characters you're allowed to use are letters (including θ) and numbers 0 to 9; this means $37+37*37=1406$ possible combinations. Of course, you should use all of the single character names first, before using the two character names. While you can technically have the same label name multiple times in a program, it is rather pointless since the calculator always goes to the first occurrence of the label.

You can position a Lbl command one or more lines before a Goto command to create a kind of loop structure. However, you have to provide the break-

```
PROGRAM: EXAMPLE
:C1rHome
:Lbl A
:Disp "HA! HA!
:Goto A
```

Command Summary

Jumps to the Lbl instruction with the specified name, and continues running the program from there.

Command Syntax

`Goto name`

...

`Lbl name`

Menu Location

While editing a program, press:

out code, since it isn't built-in. An If conditional is easiest, but if there is no code that ends the branching, then program execution will continue indefinitely, until you manually exit it (by pressing the ON key).

```
:Lbl A  
:  
:If <exit condition>  
:Goto A // this line is skipped
```

Although the Goto command may seem like a good alternative to loops, it should be avoided whenever possible, which is especially important when you are first planning a program. This is because it has several serious drawbacks associated with it:

- It is quite slow, and gets slower the further the Lbl is in your program.
- It makes reading code (your own, or someone else's) much more confusing.
- In most cases, If, For(, While, or Repeat can be used instead, saving space and improving speed.
- Using a Goto to exit any block of code requiring an End command causes a memory leak, which will not be usable until the program finishes running or executes a Return command, and which will slow down your program down. See below for ways to fix this.

The Goto command isn't all bad, however, and is actually useful when a loop isn't practical and when something only happens once or twice (see below for examples). Just remember that you should never use Goto to repeat a block of code several times. Use For(, Repeat, or While instead.

Fixing Memory Leaks

One of the simplest memory leaks that occurs is using branching to exit out of a loop when a certain condition of an If conditional is true. If the loop is an infinite loop (i.e., Repeat 0 or While 1), you should take the condition from the If conditional and place it as the condition of the loop. This allows you to remove the branching, since it is now unnecessary.

```
:Repeat 0  
:getKey→B  
:If B:Goto A  
:End:Lbl A  
Make Loop Condition  
:Repeat B  
:getKey→B  
:End
```

Of course, the only reason that this memory leak fix is possible is because of the If conditional (since the If conditional doesn't need a closing End command). When dealing with a complex If conditional, you will have to rework the conditionals so the branching has its own If conditional. Depending on how many commands there are in the conditionals, you might be able to just use an If conditional or you might need to use an If-Then conditional.

```
:If B:Then
```

1. PRGM to enter the PRGM menu.
2. 0 to choose Goto, or use arrows.
3. 9 to choose Lbl, or use arrows.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

```
:Disp "Hello
:Goto A
:End
Separate Into Conditionals
:If B:Disp "Hello
:If B:Goto A
```

This memory leak fix will work most of the time, but it isn't applicable when one of the values of the variables in the condition is changed by one of the commands inside the condition. The way to get around this is by using another variable for the If conditional that the branching uses. You initialize the variable to zero, assign the variable whatever value you want in the conditional, and then check to see if the variable is equal to that value in the branching conditional.

```
:If A=1:Then
:3→A:4→B
:Goto A
:End
Use Another Variable
:Delvar C
:If A=1:Then
:3→A:4→B:π→C
:End
:If C=π
:Goto A
```

Advanced Uses

If your program requires cleanup after it finishes, but it can exit from several different places, use Goto and place a Lbl at that point. This saves memory over repeating the cleanup code every time you exit. The usual considerations about Goto don't apply here: since you're exiting the program, all memory leaks will be gone anyway, and speed isn't much of an issue for something that only gets done once.

The code looks something like this:

```
:If K=45:Goto Q //user pressed CLEAR
...
:If L:Goto Q // game over
...
:Lbl Q
:DelVar L1ClrHome
```

A common situation in programs is when a decision has to be made about where the program execution should go next. The obvious approach would be to use the value of a variable as the label name (i.e., something like Goto A, with A being a variable), but that doesn't work because the calculator doesn't interpret the label as a variable. So, the next best approach is to use If conditionals with the different values of the variable:

```
:If not(A:Goto 0
:If A=1:Goto 1
:If A=2:Goto 2
```

Another possible use for Goto is in program protection to break a program with an error without letting the user see where it happened. If the label that you want to Goto doesn't exist, you'll get a ERR: LABEL error, which doesn't provide a 2:Goto option. So, all you have to do is Goto a label that you know doesn't exist.

Error Conditions

- ERR:INVALID occurs if this statement is used outside a program.
- ERR:LABEL is thrown if the corresponding label doesn't exist.

Related Commands

- Repeat
- While
- Menu(
- If

See Also

- Program Cleanup
- Program Protection

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/goto>

The GraphStyle(Command

The GraphStyle(command allows you to set the graphing style of an equation (line, thick line, dotted line, etc.) from within a program.

Its first argument, *equation #*, is the number of the equation whose graphing style you want to change - this depends on the mode you're in. For example, if you wanted to change the graphing style of Y_1 , you would need to be in function mode and use the value 1 for this argument. If you wanted to change the graphing style of r_4 , you would need to be in polar mode and use the value 4.

The second argument is a number from 1 to 7, which translates to a graphing style as follows:

- 1 - a normal line, usually the default graph style.
- 2 - a thick line (three pixels wide).
- 3 - a line, with everything above it shaded (only valid in function mode).
- 4 - a line, with everything below it shaded (only valid in function mode).
- 5 - a path: a line, with a ball moving along it as it is graphed (not valid in sequential mode).
- 6 - animated: a ball moving along the graph



Command Summary

Sets the graphing style of a graphing equation in the current mode.

Command Syntax

`GraphStyle(equation #, style #)`

Menu Location

While editing a program, press:

1. PRGM to access the programming menu.
2. ALPHA H to select

(not valid in sequential mode).

- 7 - a dotted line.

Compare this to the effect of Connected or Dot mode. When either of these modes is set, all equations, from all graphing modes, are reverted to line style or dotted line style respectively; furthermore, it becomes the default graph style and clearing an equation will revert it to this graph style. The GraphStyle(command simply overrides these modes temporarily.

Advanced

In shading modes (3 and 4), the shading style cycles as follows:

- The first function graphed shades using vertical lines one pixel apart
- The second function shades using horizontal lines one pixel apart
- The third function shades using negatively sloping diagonal lines, two pixels apart.
- The fourth function shades using positively sloping diagonal lines, two pixels apart.
- After that, functions will cycle through these four styles in that order.

Error Conditions

- **ERR:DOMAIN** if the *equation #* is not a valid equation number in this mode, or if *style #* is not an integer 1-7.
- **ERR:INVALID** if the graphing style chosen is not valid for the current graphing mode.

Related Commands

- FnOn
- FnOff
- Connected
- Dot

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/graphstyle>

The GridOff Command

The GridOff command disables the grid on the graph screen. This is the default setting. Use GridOn to enable the grid.

Related Commands

- GridOn
- AxesOn
- AxesOff

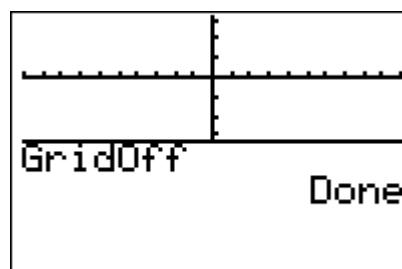
`GraphStyle(`, or use arrows.

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes



Command Summary

Disables the grid on the graph screen.

Command Syntax

GridOff

Menu Location

Press:

1. 2nd FORMAT to access the graph format menu.
2. Use arrows and ENTER to select GridOff.

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/gridoff>

The GridOn Command

The GridOn command enables a grid on the graph screen (you can disable it again with the [GridOff](#) command). How fine or coarse the grid is depends on the [Xscl](#) and [Yscl](#) variables. Drawing the grid just involves plotting points all the points of the form $(A \times Xscl, B \times Yscl)$ that are in the graphing window.

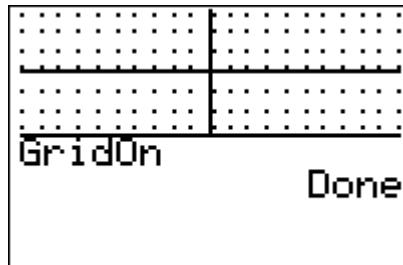
Advanced Uses

GridOn can be used to shade the entire screen if [Xscl](#) and [Yscl](#) are small enough that the points on the grid are one pixel apart:

```
:ΔX→Xscl  
:ΔY→Yscl  
:GridOn
```

This is one of the shortest ways to shade the screen, although [Shade\(](#) can be used for a (usually) even shorter way. However, using GridOn is also very slow: the fastest way involves the [Horizontal](#) or the [Vertical](#) commands in a [For\(](#) loop.

You could also use GridOn to draw the playing grid for a Dots and Boxes game.



Command Summary

Enables the grid on the graph screen.

Command Syntax

GridOn

Menu Location

Press:

1. 2nd FORMAT to access the graph format menu.
2. Use arrows and ENTER to select GridOn.

Calculator Compatibility

Related Commands

- [GridOff](#)
- [AxesOn](#)
- [AxesOff](#)

TI-83/84/+/SE

Token Size

2 bytes

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/gridon>

The G-T Commnad

G-T puts the calculator into "Graph-Table" mode: this mode shows the home screen at full size, but the graph screen and table will be displayed together, each taking up half the screen (divided vertically).

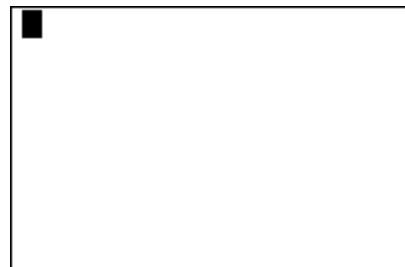
G-T is usually used at the beginning of a program to ensure that the screen mode is G-T , for programs such as math programs that want to demonstrate the thinking step-by-step.

```
:G-T
```

With OS version 2.30 (on the TI-84+ and TI-84+ SE calculators), G-T mode can be used with stat plots as well.

Related Commands

- [Full](#)
- [Horiz](#)



Command Summary

Sets the screen mode to G-T.

Command Syntax

G-T

Menu Location

In the program editor,

1. Press [MODE] for the mode menu
2. Press [DOWN] seven times (for the split screen commands)
3. Press [RIGHT] twice to select G-T
4. Press [ENTER] to insert it

This command can be used on the home screen, but must be selected from the catalog.

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/g-t>

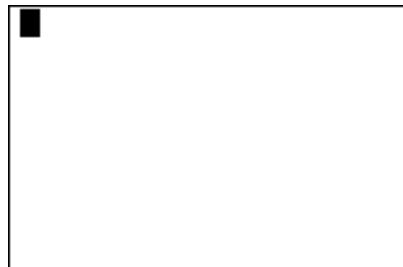
The Horiz Command

Horiz is usually at the beginning of a program. It is used at the beginning to ensure that the screen mode is Horiz, for programs such as Hangman that want to use Input but also have the graph screen shown.

```
:Horiz
```

Related Commands

- [Full](#)
- [G-T](#)



Command Summary

Sets the screen mode to Horiz.

Command Syntax

Horiz

Menu Location

In the program editor,

1. Press [MODE]
2. Press [DOWN] seven times
3. Press [RIGHT]
4. Press [ENTER] to insert Horiz

Calculator Compatibility

TI-83/84/+/SE

Token Size

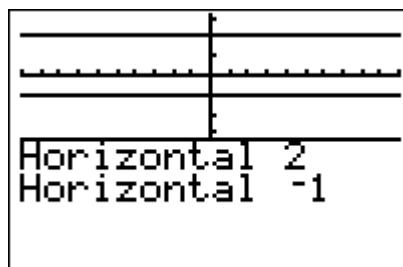
1 byte

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/horiz>

The Horizontal Command

Horizontal Y draws a vertical line from the left of the graph screen to the right at Y. Horizontal is usually only used to replace a line that stretches the entire length of the graph screen, along with its counterpart [Vertical](#).

Horizontal is affected by the window settings, unlike the [Pxl-](#) commands.



:Horizontal 5

Advanced Uses

One of the fastest ways to make the entire screen black is by drawing horizontal lines from the bottom of the screen to the top.

```
:For (A,Ymin,Ymax,ΔY  
:Horizontal A  
:End
```

Related Commands

- [Line\(](#)
- [Vertical](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/vertical>

The i Command

The *i* symbol is short for $\sqrt{-1}$, and is used for complex numbers in algebra and complex analysis. On the calculator, entering *i* will not cause an error, even in Real mode, but operations that result in a complex number (such as taking the square root of a negative number) will. If you're dealing with complex numbers, then, it's best to switch to a+bi or re^θi mode.

Advanced Uses

By using *i* in a calculation, the calculator switches to complex number mode to do it, even if in Real mode. So $\sqrt{-1}$ will throw an ERR:NONREAL ANS, but $\sqrt{0i-1}$ will not (even though it's the same number). This can be used to force calculations to be done using complex numbers regardless of the mode setting — usually by adding or subtracting $0i$, although more clever ways can be found.

A good example of this technique is our Quadratic Formula routine.

Command Summary

Draws a horizontal line on the graph screen.

Command Syntax

Horizontal Y

Menu Location

In the program editor:

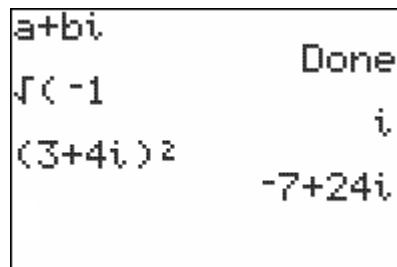
1. 2nd DRAW to enter the draw menu
2. 3 to insert the Horizontal command, or use arrows and ENTER.

Calculator Compatibility

TI-83/84/+SE

Token Size

1 byte



Command Summary

The mathematical symbol *i*, short for $\sqrt{-1}$.

Command Syntax

i

To enter a complex number:

real-part+imag-part i

Menu Location

Related Commands

- π
- e
- Real, a+bi, and re θ i

See Also

- Quadratic Formula

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/i>

The identity(Command

The identity(command generates an identity matrix: that is, a matrix [B] such that for any other matrix [A], $[A]^*[B]=[A]$ (if [A] is the right size to make the multiplication valid).

The identity matrix is square (that is, the row dimension equals the column dimension); all of its elements are 0 except for the elements along the main diagonal (the diagonal going from top left to bottom right).

The command itself takes one argument: the size of the matrix, used for both row and column size, that is, identity(n) creates an n by n matrix.

```
:dim([A]
:identity(Ans(2→[B]
:[A][B]=[A] // should always return
```

```
identity(4
[[1 0 0 0]
 [0 1 0 0]
 [0 0 1 0]
 [0 0 0 1]]
```

Command Summary

Creates an n by n identity matrix.

Command Syntax

identity(n)

Menu Location

Press:

1. MATRX (on the 83) or 2ND MATRX (83+ or higher) to access the matrix menu.
2. LEFT to access the MATH submenu.
3. 5 to select identity(, or use arrows.

Calculator Compatibility

TI-83/84/+SE

Token Size

1 byte

Optimization

The identity(command can be used as a quick way to create an empty square matrix: 0identity(n) will create an n by n matrix containing only 0 as an element. This is faster and smaller than the dim(and Fill(commands used for the same purpose:

```
:{5,5→dim([A]
:Fill(0,[A]
can be
:0identity(5→[A]
```

Press 2nd *i* to paste *i*.

Calculator Compatibility

TI-83/84/+SE

Token Size

1 byte

Error Conditions

- **ERR:INVALID DIM** occurs if the size is not an integer 1-99. In practice, however, even identity(37 requires more free RAM than the calculator ever has.

Related Commands

- [det\(](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/identity>

The If Command

The If command is crucial to most programs. It allows you to check if some condition is true, and then follow up on that by doing something if that check is true. Advanced uses of the If command allow you to have an alternate block of commands to do if the check turns out to be false. The very simplest form of the command is quite easy to understand:

```
:If condition  
:statement
```

When the calculator gets to that point in your program, it will check to see if the condition (which may be something like $2+2=4$, or $A=5$, or $\text{pxl-Test}(R,C)$, or even more complicated checks) is true. If it is, then the calculator runs the statement beneath the If, otherwise, it gets skipped. This explanation may sound complex, but in reality, If commands are so simple that you can just read most code as though it were English and understand it.

Using Then, Else, and End

Often, you want more than one statement to depend on the same condition. While you could use many If commands with that condition, one after the other, that would get long and complicated. Fortunately, TI-Basic has a solution:

```
:If condition  
:Then  
:one or more statements  
:End
```

```
PROGRAM:EXAMPLE  
:Input A  
:If A:Then  
:Disp "TRUE  
:Else  
:Disp "FALSE  
:End
```

Command Summary

Checks if a condition is true, and if it is, runs an optional statement or group of statements.

Command Syntax

If *condition*
statement

If *condition*
Then
one or more statements
End

If *condition*
Then
statement(s) to run if condition is true
Else
statement(s) to run otherwise
End

Menu Location

While editing a program, press:

1. PRGM to enter the PRGM menu
2. ENTER or 1 to choose If

Not only does this solve the problem, but it's also faster: when the condition turns out to be false, using Then increases the speed of skipping the statement(s) greatly. The trade-off, of course, is size: if you only need to work with one command to depend on the condition, then a simple If is smaller.

The next tier of complication is the Else command (note that you can only use it if you're already using Then):

```
:If condition  
:Then  
:statements if condition is true  
:Else  
:statements if condition is false  
:End
```

This provides an alternative path of running the program if the condition turns out to be false.

Advanced Uses

Each time the program enters an If-Then block, the calculator uses $35 + (\text{size of the condition})$ bytes of memory to keep track of this. This memory is given back to you as soon as the program reaches End. This isn't really a problem unless you're low on RAM, or have a lot of nested If-Then statements. However, if you use Goto to jump out of such a statement, you lose those bytes for as long as the program is running — and if you keep doing this, you might easily run out of memory, resulting in ERR:MEMORY.

Optimization

The "condition" that the If statements look for is just a number, because TI-83 series calculators don't really have specific 'true' and 'false' values. As far as the If statement is concerned, a value of 0 is false, and any other value is true. This allows for a lot of optimizations:

```
:If A≠0  
:Disp "A IS NOT 0  
can be  
:If A  
:Disp "A IS NOT 0
```

While the If command is effective, you can typically replace it with a piecewise expression when the nested statement changes a variable. Piecewise expressions function on the same 'true' or 'false' principle that the If command does, and they are not only usually smaller but more importantly faster, so they are a worthwhile alternative.

```
:If A=B  
:C+2→C  
can be  
:C+2(A=B→C)
```

Calculator Compatibility

TI-83/84/+SE

Token Size

1 byte

Code Timings

The [code timings](#) page shows that If statements with no Then and a false condition are susceptible to being greatly slowed down in For(loops that don't close a parenthesis. Specifically:

```
:For(I,1,2000)  
:If 0:  
:End
```

Takes 9 bars and 7 pixels.

```
:For(I,1,2000)  
:If 0:  
:End
```

takes 190 bars and 0 pixels — almost 20 times as long!

This is such a significant effect that you should watch out for it even if the condition is mostly true.

Error Conditions

- **ERR:DATA TYPE** occurs if the parameter is complex, even if it's complex in a silly way like 0i.
- **ERR:INVALID** occurs if this statement is used outside a program.
- **ERR:SYNTAX** occurs if an If is the last statement in the program, or the last except for one empty line.

Related Commands

- [For\(](#)
- [While](#)
- [Repeat](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/if>

The If Command

The If command is the most basic tool in programming. The idea is simple: the line after If is skipped if the condition given to If is false.

In the code below, if x really is 4, then the check $x=4$ is true, so the program will display "x equals 4". If x is not 4, then the check will be false, so Text "x equals 4" will be skipped. Nothing will be displayed.



Command Summary

```
:If x=4  
:Text "x equals 4"
```

An alternative syntax allows multiple statements to be under the control of one If. In the following example, both true→xisfour and Text "x equals 4" depend on the condition x=4.

```
:If x=4 Then  
:true→xisfour  
:Text "x equals 4"  
:EndIf
```

Using the **Else** command, an alternative can be presented if the condition is false. In the following example, "x is 4" will be displayed if x=4, and "x is not 4" otherwise.

```
:If x=4 Then  
:Text "x is 4"  
:Else  
:Text "x is not 4"  
:EndIf
```

Finally, using **ElseIf**, you can check for multiple conditions (the same effect can be accomplished using several If-Then..Else..EndIf statements, but this way is less cumbersome). A final Else, as a catch-all if no condition was met, is optional. For example:

```
:If x=4 Then  
:Text "x is 4"  
:ElseIf x=5 Then  
:Text "x is 5"  
:ElseIf x=6 Then  
:Text "x is 6"  
:Else  
:Text "x is neither 4, 5, nor 6"  
:EndIf
```

What kind of conditions are possible? Any command that returns a logical value — true or false — is acceptable. This includes:

- Relational operators: `=`, `≠`, `≥`, `≤`, `<`, and `≤`
- Logical operators: `and`, `or`, `xor`, `not`
- Any advanced test command: `pxITest()`, `isPrime()`, and others.
- A variable that contains one of the values true or false, or a function that returns them.

Of course, these can also be combined: for example, `isPrime(x)` and `x≠2` is a valid condition.

Sets a condition for a line or several lines to be executed.

Command Syntax

If *condition*
:statement

```
:If condition Then  
...  
:EndIf
```

Menu Location

From the program editor toolbar:

1. Choose F2 - Control
2. Press 1 to paste If

Calculator Compatibility

This command works on all calculators.

Token Size

3 bytes total:

- 0xE3 ('extra command' tag)
- 0x3A or 0x3B (command identifier)
- 0xE5 ('end of arguments' tag)

Optimization

Use If without a Then or EndIf for only one command; use Then and EndIf otherwise.

In addition, the [when\(\)](#) command can often replace If.

Error Conditions

- **20 - A test did not resolve to TRUE or FALSE** occurs if the condition is indeterminate, or the wrong data type.
- **730 - Missing start or end of block syntax** occurs if the If-Thens and EndIfs don't match up correctly.
- **740 - Missing Then in the If..EndIf block** occurs if a Then is missing.

Related Commands

- [when\(\)](#)
- [While](#)
- [Loop](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/68k:if>

The imag(Command

imag(z) returns the imaginary part of the complex number z. If z is represented as $x+iy$ where x and y are both real, imag(z) returns y. Also works on a list of complex numbers.

```
imag(3+4i)  
4  
  
imag({3+4i,-2i,17})  
{4,-2,0}
```

imag(3+4i)	4
imag(3-4i)	-4
imag(15)	0

Command Summary

Returns the imaginary part of a complex number.

Command Syntax

imag(*value*)

Menu Location

Press:

1. MATH to access the [math](#) menu.
2. RIGHT, RIGHT to access the CPX (complex) submenu.
3. 3 to select imag(), or use

Related Commands

- [real\(\)](#)
- [abs\(\)](#)
- [angle\(\)](#)
- [conj\(\)](#)

arrows.

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/imag>

The IndpntAsk Command

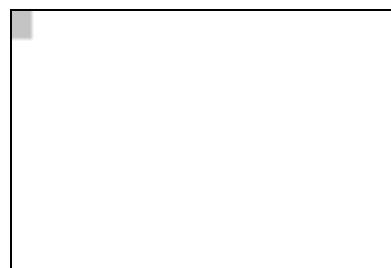
With the IndpntAsk setting, the independent variable (X , T , θ , or n depending on graphing mode) will not be calculated automatically in the table. Instead, when looking at the table, you must select an entry in the independent variable column, press ENTER, and enter a value. The values entered will also be stored to the TblInput list.

(To access the table, press 2nd TABLE, or use the DispTable command in a program)

The alternative, IndpntAuto, fills in several values starting at TblStart and increasing by ΔTbl , and makes the table scrollable (up and down).

Related Commands

- IndpntAuto
- DependAuto
- DependAsk
- DispTable



Command Summary

Doesn't automatically fill in table values for the independent variable.

Command Syntax

IndpntAsk

Menu Location

Press:

1. 2nd TBLSET to access the table settings menu.
2. Use arrows and ENTER to select Ask in the Indpnt: line.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/indpntask>

The IndpntAuto Command

The IndpntAuto setting sets the independent variable (X, T, θ , or n depending on graphing mode) to be filled in automatically in the table (which is accessible by pressing 2nd TABLE, or from a program with the DispTable command).

The values which will be filled in start at the value TblStart and increment by Δ Tbl (which can be negative, but not 0). They will also be stored in the list TblInput. All these variables can be accessed through the VARS|6:Table... menu; TblStart and Δ Tbl can also be edited in the 2nd TBLSET menu.

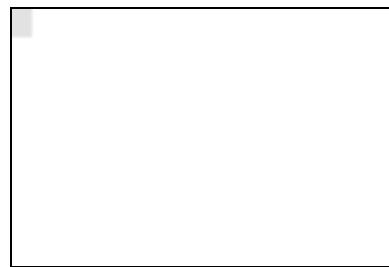
The other possibility for this setting is IndpntAsk - if that setting is turned on, you must scroll to the corresponding row in the independent variable column, and enter a value.

Error Conditions

- ERR:DOMAIN is thrown if Δ Tbl=0.

Related Commands

- IndpntAsk
- DependAuto
- DependAsk
- DispTable



Command Summary

Automatically fills in the table values for the independent variable.

Command Syntax

IndpntAuto

Menu Location

Press:

1. 2nd TBLSET to access the table settings.
2. Use arrows to select Auto in the Indpnt line to select IndpntAuto.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/indpntauto>

The Input Command

The Input command is the other way of getting user input on the home screen (getting user input on the graph screen is only possible with the getKey command). The Input command asks the user to enter a value for a variable (only one variable can be inputted at a time), waiting until the user enters a value and then presses ENTER. It does not display what variable the user is being asked for, but instead just displays a question mark (?).

PROGRAM: EXAMPLE
:Input A
:Disp "A=",A

Because just displaying a question mark on the screen does not really tell the user what to enter for input or what the input will be used for, the Input command has an optional text message that can be either text or a string variable that will be displayed alongside the input.

Only the first sixteen characters of the text message will be shown on the screen (because of the screen dimensions), so the text message should be kept as short as possible (a good goal is twelve characters or less). This is so the value the user inputs can fit on the same line as the text. In the case that the value is too long, it will wrap around to the next line.

```
PROGRAM: INPUT  
:"Fruit  
:Input "Best "+Ans,Str1  
:Input "Worst "+Ans,Str2  
:Disp "That's "+Ans+"astic!"
```

Input can be used to display every variable just before it requests user input, but some of the variables have to be entered in a certain way. If the variable is a string or a $Y=$ function, the user must put quotes ("") around the value or expression. The user must also put curly braces ({}) around lists with the list elements separated by commas, and square brackets ([]) around matrices with the matrix elements separated by commas and each row individually wrapped with square brackets.

Advanced Uses

When you just use the Input command by itself (without any arguments), the graph screen will be shown and the user can move the cursor around. Program execution will then pause until the user presses ENTER, at which time the coordinates of the cursor will be stored to the respective variables (R and θ for PolarGC format, otherwise X and Y).

If a text message is longer than twelve characters or you want to give the user plenty of space to enter a value, you can put a Disp command before the Input command. You break the text message up and display it in parts. The Input command will be displayed one line lower, though, because the Disp command automatically creates a new line.

```
:Disp "What is your"  
:Input "Name",Str0
```

Normally you can't get a quote character into a string (because quotes are used to identify the beginning and end of the string), but the Input command actually allows the user to enter a quote character ("") as part of a string. This works without problems, and the quote can even be accessed by the user afterwards.

Because a user-defined list variable doesn't need the L prefixed character before it when

Command Summary

Prompts the user to enter a value and then stores the value to the variable.

Displays the graph screen and then the user can move around the cursor.

Command Syntax

Input

Input ["Text"],variable

Menu Location

While editing a program press:

1. PRGM to enter the PRGM menu
2. RIGHT to enter the I/O menu
3. 1 to choose Input

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

referring to the list, you may be only asking the user to input a simple real variable but a list would also be allowed. There is nothing you can really do about this problem, except including the `L` prefixed character when wanting a list inputted and trying to limit your use of `Input` and `Prompt`.

```
:Input A  
should be  
:Input LA
```

Optimizations

When you are just using the text message to tell the user what the variable being stored to is, you should use the `Prompt` command instead. And, if there is a list of `Input` commands following the same pattern, you can reduce them to just one `Prompt` command.

```
:Input "A",A  
:Input "B",B  
Replace with Prompt  
:Prompt A,B
```

Error Conditions

- [ERR:INVALID](#) occurs if this statement is used outside a program.

Related Commands

- [Prompt](#)
- [getKey](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/input>

The `inString(` Command

The `inString(` command searches a string for occurrences of a smaller string (similar to the Find feature on a computer), and returns the first such occurrence.

The *source string* is the string you want to search through; the *search string* is the substring you want to find. `inString(` will return the index of the first letter of the first occurrence of the search string found, or 0 if the search string is not present. For example:

```
:inString("TI-BASIC", "BASIC  
4  
:inString("TI-BASIC", "TI  
1
```

```
inString("CAT DO  
G", "DOG  
5  
inString("CAT DO  
G DOG", "FISH  
0
```

Command Summary

Finds the first occurrence of a search string in a larger string.

Command Syntax

```
:inString("TI-BASIC", "I  
2  
:inString("TI-BASIC", "ELEPHANT  
0
```

You can also provide the optional *starting point* argument, 1 by default, that will tell the command where it should start looking. If you provide a value higher than 1 here, the command will skip the beginning of the string. This can be used to find where the search string occurs past the first occurrence. For example:

```
:inString("TI-BASIC", "I  
2  
:inString("TI-BASIC", "I", 2  
2  
:inString("TI-BASIC", "I", 3  
7
```

`inString(source string, search string, starting point)`

Menu Location

This command can only be found in the Catalog. Press:

1. 2nd CATALOG to access the command catalog
2. I to skip to command starting with I
3. scroll down to find `inString()` and select it

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

Advanced Uses

You can use `inString()` to convert a character to a number. For example:

```
:inString("ABCDEFGHIJKLMNPQRSTUVWXYZ", Str1→N
```

Assuming `Str1` is one character long and contains a capital letter, `N` will hold a value of 1..26 that corresponds to that letter. This value can then be stored in a real number, list, or matrix, where a character of a string couldn't be stored. To get the character value of the number, you can use the `sub()` command:

```
:sub("ABCDEFGHIJKLMNPQRSTUVWXYZ", N, 1→Str1
```

Using the *starting point* argument of `inString()`, you can write a routine to return all occurrences of the search string in the source string:

```
:0→dim(L1  
:inString(Str0,Str1  
:Repeat not(Ans  
:Ans→L1(1+dim(L1  
:inString(Str0,Str1,Ans+1  
:End
```

If the search string is not found, this routine will return {0} in `L1`. If it is found, the result will be a list of all the places the string was found.

Optimization

The `inString(` command can replace checking if a string is one of a number of values. Just put all the values in a string, one after the other, and try to find the string to be checked in the string of those values:

```
:If Str1=".," or Str1=","  
can be  
:If inString(".," ,Str1
```

Be careful, because if `Str1` were `".,"` in the above example, this would also be treated like `"."` or `"."`. If this is a problem, you can separate the values you want to check for by a character you know can't be in the string:

```
:If Str1="HELLO" or Str1="HI  
can be  
:If inString("HELLO,HI",Str1
```

(assuming a comma would never be in `Str1`, and words like "HELL" or "I" are also impossible)

Error Conditions

- **ERR:DOMAIN** is thrown if *starting point* is not a positive integer (it's okay, though, if it's bigger than the length of the string).

Related Commands

- [expr\(](#)
- [length\(](#)
- [sub\(](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/instring>

The `int(` Command

`int(value)` is the [floor function](#). It returns the greatest integer less than or equal to `value`. Also works on complex numbers, lists and matrices.

```
int(5.32)           5  
int(4/5)            0  
int(-5.32)          -6  
int(-4/5)           -1
```

```
int(3.16)           3  
int(-3.16)          -4  
int({3.5,3-1.5i})   {3 3-2i}
```

Command Summary

Rounds a value down to the nearest integer.

Command Syntax

The difference between [iPart\(](#) and `int(` is subtle, and

many people aren't even aware of it, but it exists. Whereas `iPart()` always truncates its parameters, simply removing the integer part, `int()` always rounds down. This means that they return the same answers for positive numbers, but `int()` will return an answer 1 less than `iPart()` for (non-integer) negative numbers. For example, `iPart(-5.32)` is -5, while `int(-5.32)` is -6.

Most of the time, however, you're dealing with only positive numbers anyway. In this case, the decision to use `iPart()` or `int()` is mostly a matter of preference - some people only use `int()` because it is shorter, some people use `iPart()` when there is a corresponding `fPart()` taken. However, see the Command Timings section.

Advanced Uses

`int()`, along with `iPart()` and `fPart()`, can be used for integer compression.

Command Timings

The following table compares the speeds of `int` and `iPart()`. Each command was timed over 2000 iterations to find a noticeable difference.

Format	Bars	Pixels	Total
<code>iPart(1)</code>	10	1	81
<code>iPart(1.643759)</code>	10	1	81
<code>int(1)</code>	8	7	71
<code>int(1.643759)</code>	10	2	82

Conclusion: Unless there are 6 or more decimals, you should consider using `int()` because of its speed, but with a lot of decimals, `iPart()` stays the same so it goes faster.

Related Commands

- `iPart()`
- `fPart()`
- `round()`

See Also

- [Compression](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/int>

`int(value)`

Menu Location

Press:

1. MATH to access the math menu.
2. RIGHT to access the NUM submenu.
3. 5 to select `int()`, or use arrows.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

The `intDiv()` Command

The integer division command, `intDiv(a,b)` returns the whole number portion of a/b : this is equal to `iPart(a/b)`. Although this operation is most useful for dividing whole numbers, this definition works for any number, whole or decimal, real or complex.

```
intDiv(125,3)
41
intDiv(-125,3)
-41
intDiv(125,π)
39
```

Advanced Uses

The `intDiv()` command also works for lists and matrices. Used with a list or matrix and a number, `intDiv()` is applied to the number paired with every element of the list or matrix. Used with two lists or two matrices, which must match in size, `intDiv()` is applied to matching elements of the list or matrix.

Use `intDiv()` and `remain()` for the quotient and remainder results of long division, respectively.

Optimization

Constructions like `iPart(a/b)` should be replaced with `intDiv(a,b)`: this is smaller and faster.

Error Conditions

- `240 - Dimension mismatch` occurs if two list or matrix arguments don't match in size.

Division by zero does not throw an error; an undefined value is returned instead.

Related Commands

- `/`
- `remain()`
- `iPart()`

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/68k:intdiv>

The `invNorm(` Command

`invNorm(` is the inverse of the cumulative normal distribution function: given a probability, it will give you a z-score with that tail probability. The

F1+ Tools	F2+ Algebra	F3+ Calc	F4+ Other	F5 Prgm	F6+ Cln Up
■	intDiv(125,3)	41			
■	intDiv(-125,3)	-41			
■	intDiv(-125,π)	-39			
■	intDiv(3+i,2·i)	-i			
	intDiv(3+i,2·i)				
MAIN	RAD AUTO	FUNC	4/30		

Command Summary

Returns the whole number part of a division.

Command Syntax

`intDiv(dividend,divisor)`

Calculator Compatibility

This command works on all calculators.

Token Size

1 byte total:

- 0xA5 (command identifier)

<code>invNorm(.5</code>	<code>)</code>
-------------------------	----------------

probability argument of `invNorm()` is between 0 and 1; 0 will give -1E99 instead of negative infinity, and 1 will give 1E99 instead of positive infinity

There are two ways to use `invNorm()`. With three arguments, the inverse of the cumulative normal distribution for a probability with specified mean and standard deviation is calculated. With one argument, the standard normal distribution is assumed (zero mean and unit standard deviation). For example:

```
for the standard normal distribution  
:invNorm(.975)  
  
for the normal distribution with mean  
:invNorm(.975,10,2.5)
```

```
invNorm(.84  
.9944578907  
invNorm(.975  
1.959963986
```

Command Summary

Calculates the inverse of the cumulative normal distribution function.

Command Syntax

`invNorm(probability[,μ, σ])`

Menu Location

Press:

1. 2ND DISTR to access the distribution menu
2. 3 to select `invNorm()`, or use arrows.

Calculator Compatibility

TI-83/84/+SE

Token Size

2 bytes

Advanced

This is the only inverse of a probability distribution function available (at least on the TI 83/+SE calculators), so it makes sense to use it as an approximation for other distributions. Since the normal distribution is a good approximation for a binomial distribution with many trials, we can use `invNorm()` as an approximation for the nonexistent "`invBinom()`". The following code gives the number of trials out of N that will succeed with probability X if the probability of any trial succeeding is P (rounded to the nearest whole number):

```
:int(.5+invNorm(X,NP,√(NP(1-P
```

You can also use `invNorm()` to approximate the inverse of a t-distribution. Since a normal distribution is a t-distribution with infinite degrees of freedom, this will be an overestimate for probabilities below 1/2, and an underestimate for probabilities above 1/2.

Formulas

Unlike the `normalpdf()` and `normalcdf()` commands, the `invNorm()` command does not have a closed-form formula. It can however be expressed in terms of the inverse error function:

$$\text{invNorm}(p) = \sqrt{2} \operatorname{erf}^{-1}(2p - 1) \quad (1)$$

For the arbitrary normal distribution with mean μ and standard deviation σ :

$$\text{invNorm}(p, \mu, \sigma) = \mu + \sigma \text{invNorm}(p) \quad (2)$$

Related Commands

- [normalpdf\(](#)
- [normalcdf\(](#)
- [ShadeNorm\(](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/invnorm>

The invT(Command

invT(is the inverse of the cumulative Student t distribution function: given a probability p and a specified degrees of freedom df , it will return the number x such that tcdf(-E99,x,df) is equal to p

```
:invT(.95,24
      1.710882023
```

Formulas

Unlike the [tpdf\(](#) and [tcdf\(](#) commands, the invT(command does not have a closed-form formula. However, it can be expressed in terms of the inverse incomplete beta function.

For the special case $df=1$, invT(is expressible in terms of simpler functions:

$$\text{invT}(p, 1) = \tan\left(\pi\left(p - \frac{1}{2}\right)\right) \quad (1)$$

Related Commands

- [tpdf\(](#)
- [tcdf\(](#)
- [Shade_t\(](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/invt>

Command Summary

Calculates the inverse of the cumulative Student t distribution function.

Command Syntax

`invT(probability, df)`

Menu Location

Press:

1. 2ND DISTR to access the distribution menu
2. 4 to select invT(, or use arrows.

Calculator Compatibility

TI-84+/SE (OS 2.30 or greater)

Token Size

2 bytes

The iPart(Command

iPart($value$) returns the integer part of $value$. Also works on complex numbers, lists and matrices.

```
iPart(5.32)
      5
iPart(4/5)
```

<code>iPart(3.16</code>	<code>3</code>
<code>iPart(-3.16</code>	<code>-3</code>
<code>iPart({3.2, 1.7}</code>	<code>{3 13</code>

iPart(-5.32)	0
iPart(-5)	-5
iPart(-4/5)	0

The difference between `iPart(` and `int(` is subtle, and many people aren't even aware of it, but it exists. Whereas `iPart(` always truncates its parameters, simply removing the integer part, `int(` always rounds down. This means that they return the same answers for positive numbers, but `int(` will return an answer 1 less than `iPart(` for (non-integer) negative numbers. For example, `iPart(-5.32)` is -5, while `int(-5.32)` is -6.

Most of the time, however, you're dealing with only positive numbers anyway. In this case, the decision to use `iPart(` or `int(` is mostly a matter of preference - some people only use `int(` because it is shorter, some people use `iPart(` when there is a corresponding `fPart(` taken. However, see the Command Timings section.

Advanced Uses

`iPart(`, along with `fPart(` and `int(`, can be used for integer compression.

Command Timings

The following table compares the speeds of `int(` and `iPart(`. Each command was timed over 2000 iterations to find a noticeable difference.

Format	Bars	Pixels	Total
<code>iPart(1</code>	10	1	81
<code>iPart(1.643759</code>	10	1	81
<code>int(1</code>	8	7	71
<code>int(1.643759</code>	10	2	82

Conclusion: Unless there are 6 or more decimals, you should consider using `int(` because of it's speed, but with a lot of decimals, `iPart(` stays the same so it goes faster.

Related Commands

- `int(`
- `fPart(`
- `round(`

See Also

Command Summary

Returns the integer part of a value.

Command Syntax

`iPart(value)`

Menu Location

Press:

1. MATH to access the math menu.
2. RIGHT to access the NUM submenu
3. 3 to select `iPart(`, or use arrows.

Calculator Compatibility

TI-83/84/+SE

Token Size

1 byte

- Compression

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/irr>

The irr(Command

The irr(command finds the Internal Rate of Return of an investment, which is a measure of its efficiency. Its mathematical interpretation is the interest rate for which npv(will return 0 for the same cash flows.

irr(takes three arguments: an initial cash flow (CF0), a list of further cash flows (CFList), and an optional frequency list.

Advanced Uses

irr(can be used to find a root of a polynomial of any degree, give by a list of its coefficients:

```
1+.01irr(0,{list of coefficients})
```

However, this method is limited to finding roots greater than 1, and will throw an error (ERR:NO SIGN CHG or ERR:DIVIDE BY 0) if it can't find such roots. By reversing the list of coefficients and taking the reciprocal of the roots found, you could find roots less than 1, but this would still result in errors if such roots don't exist either.

Using solve(to find roots of polynomials is less efficient, but more reliable, since it doesn't throw an error unless there are no roots at all to be found.

Formulas

Solving for irr(requires solving a polynomial with degree equal to the total number of cash flows. As such, there is no general formula for calculating irr(, though numerical methods are possible for finding an approximate solution.

The polynomial associated with the calculation is:

$$\sum_{i=0}^N C_i \left(1 + \frac{\text{Irr}}{100}\right)^{N-i} = 0 \quad (1)$$

```
irr(-100,{30,35,  
40,45})  
17.09368634
```

Command Summary

Calculates the Internal Rate of Return of an investment.

Command Syntax

```
irr(CF0,CFList,[freq])
```

Menu Location

On the TI-83, press:

1. 2nd FINANCE to access the finance menu.
2. 8 to select irr(, or use arrows and ENTER.

On the TI-83+ or higher, press:

1. APPS to access the applications menu.
2. 1 or ENTER to select Finance...
3. 8 to select irr(, or use arrows and ENTER.

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

Here, Irr is the internal rate of return, N is the number of cash flows, and C_t is the t^{th} cash flow.

To the calculator, only roots for which $\text{Irr} > 0$ are considered to be viable.

Error Conditions

- **ERR:DIM MISMATCH** is thrown if the frequency list's size doesn't match the cash flow list's size.
- **ERR:DIVIDE BY 0** is thrown if the solution that is found is $\text{Irr} = 0$.
- **ERR:NO SIGN CHG** is thrown if no positive real solution is found.

Related Commands

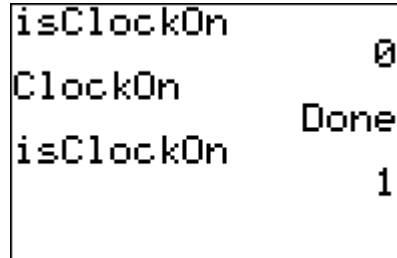
- npv(

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/irr>

The isClockOn Command

The `isClockOn` command returns whether the clock on the TI-84+/SE calculators is on or off. Based on Boolean logic, it will return 1 if it is on and 0 if it is not. You can store it to a variable for later use, or use it in conditionals and loops as part of the condition. For example, here is how you would check to see if the clock is on:

```
:If isClockOn  
:Then  
    (code if clock is on)  
:Else  
    (code if clock is off)  
:End
```



Command Summary

Returns whether the clock on the TI-84+/SE is on or off.

Command Syntax

`isClockOn`

Menu Location

This command can only be found in the catalog. Press:

1. 2nd CATALOG to enter the command catalog
2. i to skip to commands starting with I
3. Scroll down to `isClockOn` and select it

Calculator Compatibility

TI-84+/SE

Token Size

Related Commands

- ClockOff
- ClockOn

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/isclockon>

The IS>(Command

The increment and skip if greater than command — IS> — is a specialized conditional command. It is equivalent to an If conditional, except the next command will be skipped when the condition is true and it has a variable update built-in. However, it is not used very often (if anything, it is often misused as a looping command) because of its obscure name and somewhat limited application.

The IS> command takes two arguments:

- A variable, which is limited only to one of the real variables (A-Z or θ).
- A value, which can be either a number, variable, or expression (a combination of numbers and variables).

When IS> is executed it adds one to the variable (increments it by one), and compares it to the value. The next command will be skipped if the variable is greater than the value, while the next command will be executed if the variable is less than or equal to the value.

The command IS>(A,B is equivalent to the following code:

```
:A+1→A
:If A≤B
```

Here are the two main cases where the IS> command is used:

```
:7→A
:IS>(A,6
:Disp "Skipped
```

- Initializes the A variable to 7 and then compares to the value
- 7>6 is true so the display message won't be displayed

```
:1→B
:IS>(B,2
:Disp "Not Skipped
```

PROGRAM: EXAMPLE

```
:3→A
:While A<9
:IS>(A,8
:Disp A
:End
```

Command Summary

Increments a variable by 1 and skips the next command if the variable is greater than the value.

Command Syntax

*IS>(variable,value)
command*

Menu Location

While editing a program, press:

1. PRGM to enter the PRGM menu
2. A to choose IS>, or use arrows

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

- Initializes the B variable to 1 and then compares to the value
- $1>2$ is false so the display message will be displayed

Note: In addition to both of these cases, there is also the case where the variable and the value are equal to each other. This case is shown below under the 'Advanced Uses' section because it has some added background that goes with it.

Advanced Uses

When you want the skipping feature of the `IS>(` command to always occur, you just have to use the same variable for both the variable and value arguments of the command:

```
: IS>(B, B
```

An undefined error will occur if the variable and/or value doesn't exist before the `IS>(` command is used, which happens when the [DelVar](#) command is used. Consequently, you should not use `DelVar` with `IS>()`.

Similar code can be used as a substitute for $B+1 \rightarrow B$ if you don't want to change [Ans](#):

```
: IS>(B, B :
```

Note that due to the colon after the line, there will be no statement skipped, so you don't have to worry about that.

Optimization

Because the `IS>(` command has the variable update built-in, it is smaller than manually incrementing a variable by one along with using an `If` conditional.

```
: A+1→A  
can be  
: IS>(A, 0
```

The one caution about this is that if the variable is greater than the value (in this case, '0'), the next command will be skipped. If you don't want the skipping functionality, then you need to make sure that the value is never less than the variable. This is not always possible to do. Also, `IS>(` is slightly slower than its more normal counterpart.

Related to the example code given, `IS>(` should always have a command following after it (i.e., it's not the last command in a program) because it will return an error otherwise. If you have no particular code choice, just put an empty line or something meaningless.

Command Timings

Using `IS>(` to increment a variable is approximately 25% slower than using code like $X+1 \rightarrow X$. However, it is faster to use `IS>(` than to construct an `If` statement to do the same thing.

Note, however, that a quirk in the [For](#)(command (see its Optimizations section) will slow down

the IS>(command significantly if a closing parenthesis is not used for the For(statement.

Error Conditions

- ERR:INVALID occurs if this statement is used outside a program.
- ERR:UNDEFINED is thrown if the variable to be incremented is not defined.
- ERR:SYNTAX is thrown if there is no next line to skip, or if there is only one next line and it is empty.

Related Commands

- DS<(
- If

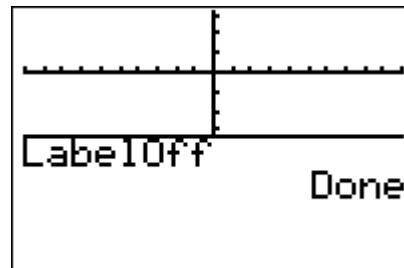
For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/is>

The LabelOff Command

The LabelOff setting disables labels on the X and Y coordinate axes. This is unnecessary if you've disabled the axes themselves, since the labels are only displayed when the axes are. To enable the labels, use the reverse setting LabelOn.

Related Commands

- LabelOn
- AxesOn
- AxesOff



Command Summary

Disables labels on the X and Y coordinate axes.

Command Syntax

LabelOff

Menu Location

Press:

1. 2nd FORMAT to access the format menu.
2. Use arrows and ENTER to select LabelOff.

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

The LabelOn Command

The LabelOn setting enables labels on the X and Y coordinate axes. If both LabelOn and AxesOn are set, the axes will be displayed with an X next to the X (horizontal) axis, and a Y next to the Y (vertical) axis. To disable these labels, use the LabelOff setting.

LabelOn and LabelOff have no effect if the coordinate axes are displayed: there's nothing to label.

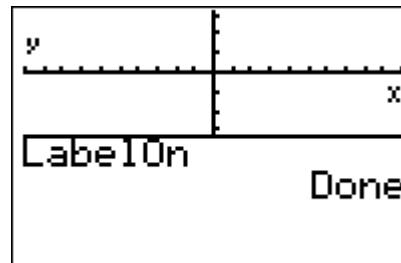
A somewhat quirky behavior of the X and Y labels is that they aren't saved by StorePic. If you save a picture of the graph screen, it records every detail of the way it looks, including equations, drawn elements, axes, grid, everything — but not the labels.

One final comment: okay, so by the way the command works we know it was once *intended* to label the axes. However, the command doesn't actually check where the axes *are*. It puts an "x" slightly above the bottom right corner, and a "y" slightly below the top left. Most of the time, including the default graphing window, that doesn't help you to distinguish the axes in the slightest. And in split-screen mode, as shown in the screenshot, they both seem to label the x and y axis. Weird.

Related Commands

- LabelOff
- AxesOn
- AxesOff

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/labelon>



Command Summary

Puts labels on the X and Y coordinate axes.

Command Syntax

LabelOn

Menu Location

Press:

1. 2nd FORMAT to access the format menu.
2. Use arrows and ENTER to select LabelOn.

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

The Lbl Command

The Lbl command is used together with the Goto command to jump (or branch) to another place in a program. When the calculator executes a Goto command, it stores the label name in memory, and then searches from the beginning of the program for the Lbl command with the supplied name. If it finds it, it continues running the program from that point;

PROGRAM: EXAMPLE
:ClrHome
:Lbl A
:Disp "HA! HA!
:Goto A

otherwise, if the label does not exist, it throws a **ERR: LABEL** error.

Label names can be either one or two characters long, and the only characters you're allowed to use are letters (including θ) and numbers 0 to 9; this means $37+37*37=1406$ possible combinations. Of course, you should use all of the single character names first, before using the two character names. While you can technically have the same label name multiple times in a program, it is rather pointless since the calculator always goes to the first occurrence of the label.

You can position a Lbl command one or more lines before a Goto command to create a kind of loop structure. However, you have to provide the break-out code, since it isn't built-in. An If conditional is easiest, but if there is no code that ends the branching, then program execution will continue indefinitely, until you manually exit it (by pressing the ON key).

```
:Lbl A  
:...  
:If <exit condition>  
:Goto A // this line is skipped
```

Although the Lbl/Goto loop structure may seem like a good alternative to loops, it should be avoided whenever possible, which is especially important when you are first planning a program. This is because it has several serious drawbacks associated with it:

- It is quite slow, and gets slower the further the Lbl is in your program.
- It makes reading code (your own, or someone else's) much more confusing.
- In most cases, If, For(, While, or Repeat can be used instead, saving space and improving speed.
- Using a Goto to exit any block of code requiring an End command causes a memory leak — around 40 bytes of memory will be rendered useless each time you do it until the program finishes running, which will also slow down your program down.

They aren't all bad, however, and are actually useful when a loop isn't practical and when something only happens once or twice. Just remember that you should never use Goto to repeat a block of code several times. Use For(, Repeat, or While instead.

Labels are also used with the Menu(command. The same considerations apply as with Goto, except that (unless you write a custom menu routine) there's no simple alternative to using labels with Menu(.

Error Conditions

- **ERR:INVALID** is thrown if this statement is used outside a program.
- **ERR:LABEL** is thrown if the corresponding label doesn't exist.

Related Commands

Command Summary

Defines a label for a particular Goto or Menu(to jump to.

Command Syntax

Lbl *name*

Menu Location

While editing a program, press:

1. PRGM to enter the PRGM menu
2. 9 to choose Lbl, or use arrows

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

- [Goto](#)
- [Menu\(](#)
- [While](#)
- [Repeat](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/lbl>

The lcm(Command

Returns the least common multiple (LCM) of two nonnegative integers; $\text{lcm}(a,b)$ is equivalent to $ab/\text{gcd}(a,b)$. Also works on lists.

```
lcm(8,6)
24
lcm({9,12},6)
{18 12}
lcm({14,12},{6,8})
{42 24}
```

<code>lcm(3,5</code>	<code>15</code>
<code>lcm(0,9</code>	<code>0</code>
<code>lcm(6,lcm(8,10</code>	<code>120</code>

Error Conditions

- **ERR:DIM MISMATCH** is thrown if the arguments are two lists that don't have the same number of elements.
- **ERR:DOMAIN** is thrown if the arguments aren't positive integers (or lists of positive integers) less than $1e12$.

Related Commands

- [gcd\(](#)

Command Summary

Finds the least common multiple of two values.

Command Syntax

`lcm(value1, value2)`

Menu Location

Press:

1. MATH to access the [math](#) menu.
2. RIGHT to access the NUM submenu.
3. 8 to select `lcm(`, or use arrows.

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/lcm>

The length(Command

This command is used to determine the length of a string. Unlike the [dim\(](#) command for lists and matrices, it cannot be used to change this length.

```
:length("HELLO  
5
```

Keep in mind that the length is measured in the number of tokens, and not the number of letters in the string. For example, although the [sin\(](#) command contains 4 characters ("s", "i", "n", and "("), it will only add 1 to the total length of a string it's in.

Advanced Uses

The code for looping over each character (technically, each token) of a string involves [length\(](#):

```
:For(I,1,length(Str1  
...  
use sub(Str1,I,1 for the Ith character  
...  
:End
```

Related Commands

- [expr\(](#)
- [inString\(](#)
- [sub\(](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/length>

The Line(Command

The [Line\(](#) command is used to draw lines at any angle, as opposed to only drawing [vertical](#) or [horizontal](#) lines. [Line\(X₁,Y₁,X₂,Y₂\)](#) will draw a line from (X₁,Y₁) to (X₂,Y₂). [Line\(](#) is affected by the window settings, although you can use a [friendly window](#) so there is no impact on the command.

```
:Line(5,5,20,3)
```

length("HELLO	5
length("sin(X	2

Command Summary

Returns the length of a string.

Command Syntax

[length\(string\)](#)

Menu Location

This command can only be found in the catalog. Press:

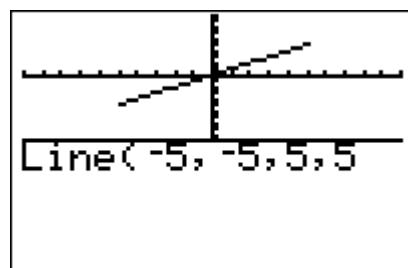
1. 2nd CATALOG to access the command catalog
2. L to skip to commands starting with L
3. scroll down to [length\(](#) and select it

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes



Command Summary

Advanced Uses

Line has an optional fifth argument. It can be any real number, but the default is one. If the fifth argument, *erase*, is something other than 0, then it will simply draw the line. If *erase* is 0, then it will erase the line.

```
:Line(5,5,20,3,0)
```

Leave off the ending argument if you are just drawing the line.

```
:Line(5,5,20,3,1)  
can be  
:Line(5,5,20,3)
```

Also, don't forget that the end number can be a formula, which is useful for movement applications and other things such as health bars where the lines drawn are constantly different. The following draws a line unless a key is pressed, in which case it will not.

```
:getKey→K  
:Line(5,5,20,3,not(K))
```

Draws a line at any angle.

Command Syntax

`Line(X1,Y1,X2,Y2[,argument])`

Menu Location

Press:

1. 2nd DRAW to enter the draw menu
2. 2 to insert the Line(token, or use arrows and ENTER.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

Command Timings

If you are drawing horizontal or vertical lines that stretch the entire graph screen, such as a border, it is better to use Vertical or Horizontal. These are smaller and are usually faster as well.

Related Commands

- Vertical
- Horizontal

See Also

- Friendly Graphing Window

Bibliography

1. full source reference

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/line>

The LinReg(a+bx) Command

The LinReg(a+bx) command is one of several that can calculate the line of best fit through a set of points (it differs from LinReg(ax+b) only in the format of its output). To use it, you must first store the points to two lists: one of the x-coordinates and one of the y-coordinates, ordered so that the Nth element of one list matches up with the Nth element of the other list. L1 and L2 are the default lists to use, and the List Editor (STAT > Edit...) is a useful window for entering the points.

In its simplest form, LinReg(a+bx) takes no arguments, and calculates a best fit line through the points in L1 and L2:

```
: {9,13,21,30,31,31,34→L1  
: {260,320,420,530,560,550,590→L2  
: LinReg(a+bx)
```

On the home screen, or as the last line of a program, this will display the equation of the line of best fit: you'll be shown the format, $y=a+bx$, and the values of a and b. It will also be stored in the RegEQ variable, but you won't be able to use this variable in a program - accessing it just pastes the equation wherever your cursor was. Finally, the statistical variables a, b, r, and r^2 will be set as well. These latter two variables will be displayed only if "Diagnostic Mode" is turned on (see DiagnosticOn and DiagnosticOff).

You don't have to do the regression on L1 and L2, but if you don't you'll have to enter the names of the lists after the command. For example:

```
: {9,13,21,30,31,31,34→FAT  
: {260,320,420,530,560,550,590→CALS  
: LinReg(a+bx) ↴FAT, ↴CALS
```

You can attach frequencies to points, for when a point occurs more than once, by supplying an additional argument - the frequency list. This list does not have to contain integer frequencies. If you add a frequency list, you must supply the names of the x-list and y-list as well, even when they're L1 and L2.

Finally, you can enter an equation variable (such as Y_1) after the command, so that the line of best fit is stored to this equation automatically. This doesn't require you to supply the names of the lists, but if you do, the equation variable must come last. You can use polar, parametric, or sequential variables as well, but since the line of best fit will be in terms of X anyway, this doesn't make much sense.

An example of LinReg(a+bx) with all the optional arguments:

Command Summary

Calculates the best fit line through a set of points.

Command Syntax

LinReg(a+bx) [*x-list*, *y-list*,
[*frequency*], [*equation*]]

Menu Location

Press:

1. STAT to access the statistics menu
2. LEFT to access the CALC submenu
3. 8 to select LinReg(a+bx), or use arrows

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

```
: {9,13,21,30,31,31,34→FAT
: {260,320,420,530,560,550,590→CALS
: {2,1,1,1,2,1,1→FREQ
: LinReg(ax+b) ↵ FAT,CALS,FREQ,Y1
```

Advanced Uses (for programmers)

`LinReg(a+bx)`, along with [LinReg\(ax+b\)](#), can be used to [convert a number to a string](#).

Related Commands

- [LinReg\(ax+b\)](#)
- [LinRegTTest](#)
- [LinRegTInt](#)
- [Manual-Fit](#)
- [Med-Med](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/linreg-a-bx>

The `LinReg(ax+b)` Command

The `LinReg(ax+b)` is one of several commands that can calculate the line of best fit through a set of points. To use it, you must first store the points to two lists: one of the x-coordinates and one of the y-coordinates, ordered so that the Nth element of one list matches up with the Nth element of the other list. L1 and L2 are the default lists to use, and the List Editor (STAT > Edit...) is a useful window for entering the points.

In its simplest form, `LinReg(ax+b)` takes no arguments, and calculates a best fit line through the points in L1 and L2:

```
: {9,13,21,30,31,31,34→L1
: {260,320,420,530,560,550,590→L2
: LinReg(ax+b)
```

On the home screen, or as the last line of a program, this will display the equation of the line of best fit: you'll be shown the format, $y=ax+b$, and the values of a and b . It will also be stored in the `RegEQ` variable, but you won't be able to use this variable in a program - accessing it just pastes the equation wherever your cursor was. Finally, the statistical variables a , b , r , and r^2 will be set as well. These latter two variables will be displayed only if "Diagnostic Mode" is turned on (see [DiagnosticOn](#)

Command Summary

Calculates the best fit line through a set of points.

Command Syntax

`LinReg(ax+b) [x-list, y-list, [frequency], [equation]]`

Menu Location

Press:

1. STAT to access the statistics menu
2. LEFT to access the CALC submenu
3. 4 to select `LinReg(ax+b)`, or use arrows

Calculator Compatibility

TI-83/84/+SE

Token Size

1 byte

and [DiagnosticOff](#)).

You don't have to do the regression on L1 and L2, but if you don't you'll have to enter the names of the lists after the command. For example:

```
:{9,13,21,30,31,31,34→FAT  
:{260,320,420,530,560,550,590→CALS  
:LinReg(ax+b) ∟FAT,∟CALS
```

You can attach frequencies to points, for when a point occurs more than once, by supplying an additional argument - the frequency list. This list does not have to contain integer frequencies. If you add a frequency list, you must supply the names of the x-list and y-list as well, even when they're L1 and L2.

Finally, you can enter an equation variable (such as Y_1) after the command, so that the line of best fit is stored to this equation automatically. This doesn't require you to supply the names of the lists, but if you do, the equation variable must come last. You can use polar, parametric, or sequential variables as well, but since the line of best fit will be in terms of X anyway, this doesn't make much sense.

An example of LinReg(ax+b) with all the optional arguments:

```
:{9,13,21,30,31,31,34→FAT  
:{260,320,420,530,560,550,590→CALS  
:{2,1,1,1,2,1,1→FREQ  
:LinReg(ax+b) ∟FAT,∟CALS,∟FREQ,Y1
```

Advanced Uses (for programmers)

LinReg(ax+b), along with [LinReg\(a+bx\)](#), can be used to [convert a number to a string](#).

Related Commands

- [LinReg\(a+bx\)](#)
- [LinRegTTest](#)
- [LinRegTInt](#)
- [Manual-Fit](#)
- [Med-Med](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/linreg-ax-b>

The LinRegTInt Command

Like [LinReg\(ax+b\)](#) and similar commands, LinRegTInt finds the best fit line through a set of points. However, LinRegTInt adds another method of checking the quality of the fit, by calculating a t confidence interval for the slope b. If the confidence interval calculated contains zero, the data supplied is insufficient to conclude a linear relation between

Command Summary

Calculates the linear regression of two sets of data with a confidence interval for the slope coefficient.

the variables.

To use LinRegTInt, you must first store the points to two lists: one of the x-coordinates and one of the y-coordinates, ordered so that the Nth element of one list matches up with the Nth element of the other list. L1 and L2 are the default lists to use, and the List Editor (STAT > Edit...) is a useful window for entering the points. You do not have to do the regression on L1 and L2, but if you don't you'll have to enter the names of the lists after the command.

You can attach frequencies to points, for when a point occurs more than once, by supplying an additional argument - the frequency list. This list does not have to contain integer frequencies. If you add a frequency list, you must supply the names of the x-list and y-list as well, even when they are L1 and L2.

You can supply a confidence level probability as the fourth argument. It should be a real number between zero and one. If not supplied, the default value is .95. (95% confidence level) If you need to specify a different confidence level, you must enter the names of the lists as well, even if they're L1 and L2.

Finally, you can enter an equation variable (such as Y_1) after the command, so that the line of best fit is stored to this equation automatically. This doesn't require you to supply the names of the lists, but if you do, the equation variable must come last.

For example, both

```
:{4,5,6,7,8→L1  
:{1,2,3,3.5,4.5→L2  
:LinRegTInt
```

and

```
:{4,5,6,7,8→X  
:{1,2,3,3.5,4.5→Y  
:{1,1,1,1,1→FREQ  
:LinRegTTest LX,LY,LFREQ,.95,Y1
```

will give the following output:

```
LinRegTInt  
y=a+bx  
(.69088,1.0091)  
b=.85  
df=3  
s=.158113883  
a=-2.3  
r2=.9897260274
```

Command Syntax

LinRegTInt [*x-list*, *y-list*,
[*frequency*], [*confidence level*],
[*equation*]]

Menu Location

Press:

1. STAT to access the statistics menu
2. RIGHT to access the TESTS submenu
3. ALPHA G to select LinRegTInt, or use arrows

Calculator Compatibility

TI-84+(SE) OS 2.30 or greater

Token Size

2 bytes

r=.9948497512

(the last two lines will only appear if diagnostics have been turned on - see [DiagnosticOn](#))

- The first line shows the confidence interval containing the slope of the fitted line; as mentioned above, if the interval contains 0, it cannot be concluded that the two variables have a linear relationship. Also, the smaller the difference between the two numbers, the more precision that can be attributed to the calculated slope.
- df is the degrees of freedom, equal to the number of points minus two.
- a and b are the parameters of the equation $y=a+bx$, the regression line we've calculated
- s is the standard error about the line, a measure of the typical size of a residual (the numbers stored in LRESID). It is the square root of the sum of squares of the residuals divided by the degrees of freedom. Smaller values indicate that the points tend to be close to the fitted line, while large values indicate scattering.
- r^2 and r are respectively the coefficients of determination and correlation: a value near 1 or -1 for the former, and near 1 for the latter, indicates a good fit.

Related Commands

- [LinReg\(ax+b\)](#)
- [LinReg\(a+bx\)](#)
- [LinRegTTest](#)
- [Manual-Fit](#)
- [Med-Med](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/linregttint>

The LinRegTTest Command

Like [LinReg\(ax+b\)](#) and similar commands, LinRegTTest finds the best fit line through a set of points. However, LinRegTTest adds another method of checking the quality of the fit, by performing a t-test on the slope, testing the null hypothesis that the slope of the true best fit line is 0 (which implies the absence of correlation between the two variables, since a relation with a slope of zero means the x-variable does not affect the y-variable at all). If the p-value of the test is not low enough, then there is not enough data to assume a linear relation between the variables.

To use LinRegTTest, you must first store the points to two lists: one of the x-coordinates and one of the y-coordinates, ordered so that the Nth element of one list matches up with the Nth element of the other list. L1 and L2 are the default lists to use, and the List Editor (STAT > Edit...) is a useful window for entering the points.

In its simplest form, LinRegTTest takes no arguments, and calculates a best fit line through the points in L1 and L2:

```
LinRegTTest  
y=a+bx  
B≠0 and P≠0  
t=29.67282165  
P=8.4066188E-5  
df=3  
4a=-.2250341426
```

Command Summary

Calculates the best fit line through a set of points, then uses a significance test on the slope of the line.

Command Syntax

```
LinRegTTest [x-list, y-list,  
[frequency], [alternative], [equation]]
```

Menu Location

Press:

```
: {9,13,21,30,31,31,34→L1
: {260,320,420,530,560,550,590→L2
: LinRegTTest
```

The output will look as follows:

```
LinRegTTest
y=a+bx
 $\beta \neq 0$  and  $\rho \neq 0$ 
t=53.71561274
p=4.2285344e-8
df=5
a=145.3808831
b=13.09073265
s=5.913823968
r2=.9982701159
r=.9991346836
```

1. STAT to access the statistics menu
2. RIGHT to access the TESTS submenu
3. ALPHA E to select LinRegTTest, or use arrows

Change the last keypress to ALPHA F on a TI-84+/SE with OS 2.30 or higher.

Calculator Compatibility

TI-83/84+/SE

Token Size

2 bytes

(the last two lines will only appear if diagnostics have been turned on - see [DiagnosticOn](#))

- β and ρ : this line represents the alternative hypothesis. β is the true value of the statistic b (it is what we would get if the regression was done on the entire population, rather than a sample); ρ is the true value of the statistic r.
- t is the test statistic, used to calculate p.
- p is the probability that we'd get a correlation this strong by chance, assuming the null hypothesis that there is no actual correlation. When it's low, as here, this is evidence against the null hypothesis. Since p<.01, the data is significant on a 1% level, so we reject the null hypothesis and conclude that there is a correlation.
- df is the degrees of freedom, equal to the number of points minus two
- a and b are the parameters of the equation $y=a+bx$, the regression line we've calculated
- s is the standard error about the line, a measure of the typical size of a residual (the numbers stored in L RESID). It is the square root of the sum of squares of the residuals divided by the degrees of freedom. Smaller values indicate that the points tend to be close to the fitted line, while large values indicate scattering.
- r^2 and r are respectively the coefficients of determination and correlation: a value near 1 or -1 for the former, and near 1 for the latter, indicates a good fit.

You do not have to do the regression on L1 and L2, but if you don't you'll have to enter the names of the lists after the command. For example: (L is the small L found in the LIST>OPS menu)

```
: {9,13,21,30,31,31,34→FAT
: {260,320,420,530,560,550,590→CALS
: LinRegTTest L,FAT,L,CALS
```

You can attach frequencies to points, for when a point occurs more than once, by supplying an additional argument - the frequency list. This list does not have to contain integer frequencies. If you add a frequency list, you must supply the names of the x-list and y-list as well, even when they are L1 and L2.

You can add the *alternative* argument to change the alternative hypothesis from the default ($\beta \neq 0$ and $\rho \neq 0$). This is used when you have prior knowledge either that a negative relation is

impossible, or that a positive one is impossible. The values of the *alternative* argument are as follows:

- negative: the alternative hypothesis is $\beta < 0$ and $p < 0$ (we have prior knowledge that there can be no positive relation)
- 0: the alternative hypothesis is $\beta \neq 0$ and $p \neq 0$ (we have no prior knowledge)
- positive: the alternative hypothesis is $\beta > 0$ and $p > 0$ (we have prior knowledge that there can be no negative relation)

Obviously, if you want the alternative hypothesis to be $\beta \neq 0$ and $p \neq 0$, the default, you don't need to supply this argument. However, if you do, you must enter the names of the lists as well, even if they're L1 and L2.

Finally, you can enter an equation variable (such as Y_1) after the command, so that the line of best fit is stored to this equation automatically. This doesn't require you to supply the names of the lists, but if you do, the equation variable must come last. You can use polar, parametric, or sequential variables as well, but since the line of best fit will be in terms of X anyway, this doesn't make much sense.

An example of LinRegTTest with all the optional arguments:

```
: {9,13,21,30,31,31,34→FAT
: {260,320,420,530,560,550,590→CALS
: {2,1,1,1,2,1,1→FREQ
: LinRegTTest ↵FAT,↵CALS,↵FREQ,1,Y1
```

Related Commands

- [LinReg\(ax+b\)](#)
- [LinReg\(a+bx\)](#)
- [LinRegTInt](#)
- [Manual-Fit](#)
- [Med-Med](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/linregttest>

The List►matr(Command

The List►matr(builds a matrix by combining several list expressions, and stores it to the specified variable ([A] through [J]). Each list specifies a column of the matrix: the first list will be stored down the first (leftmost) column, the second list down the second column, and so on. For example:

```
List►matr({1,2,3},{10,20,30},{100,200)
          Done
[A]
[[1 10 100]
 [2 20 200]
 [3 30 300]]
```

```
List►matr({1,2,3
},{4,5},{6},{A})
          Done
[A]
[[1 4 6]
 [2 5 0]
 [3 0 0]]]
```

Command Summary

Builds a matrix from one or more lists.

Advanced Uses

The calculator can actually handle lists that are not the same size. It will pad zeroes to the shorter lists, until they have the same length as the longest list.

```
List►matr({1,2,3},{10},{100,200},[A]  
Done  
[A]  
[[1 10 100]  
 [2 0 200]  
 [3 0 0 ]]
```

Error Conditions

- **ERR:ARGUMENT** is thrown if there are more than 99 lists (since a matrix can be at most 99x99)
- **ERR:INVALID DIM** is thrown if one of the lists is longer than 99 elements (since a matrix can be at most 99x99)

Related Commands

- [Matr►list\(](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/list-matr>

The ln(Command

The ln(command computes the natural logarithm of a value — the exponent to which the constant e must be raised, to get that value. This makes it the inverse of the [e^\(](#) command.

ln(is a real number for all positive real values. For negative numbers, ln(is an imaginary number (so taking ln(of a negative number will cause **ERR:NONREAL ANS** to be thrown in Real mode), and of course it's a complex number for complex values. ln(is not defined at 0, even if you're in a complex mode.

Advanced Uses

Using either the ln(or the [log\(](#) command, logarithms

Command Syntax

List►matr(*list1, [list2, ...], matrix*

Menu Location

Press:

1. MATRIX (on the 83) or 2nd MATRIX (83+ or higher) to access the matrix menu
2. LEFT to access the MATH submenu
3. 9 to select List►matr(, or use arrows.

Alternatively, press:

1. 2nd LIST to access the list menu
2. LEFT to access the OPS submenu
3. 0 to select List►matr(, or use arrows

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

ln(1	0
ln(e	1
ln(e^99	99

Command Summary

Computes the (principal branch of the) natural logarithm.

Command Syntax

ln(*value*)

of any base can be calculated, using the identity:

$$\log_b x = \frac{\ln x}{\ln b} = \frac{\log_c x}{\log_c b} \quad (1)$$

So, to take the base B log of a number X, you could use either of the following equivalent ways:

```
: log(X)/log(B)
```

```
: ln(X)/ln(B)
```

This is the exponent to which B must be raised, to get X.

Error Conditions

- [ERR:DOMAIN](#) when calculating $\ln(0)$.
- [ERR:NONREAL ANS](#) if taking $\ln($ of a negative number in Real mode.

Related Commands

- [e](#)
- [e^\(](#)
- [log\(](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/ln>

The LnReg Command

LnReg tries to fit a logarithmic curve ($y=a+b\ln x$) through a set of points. To use it, you must first store the points to two lists: one of the x-coordinates and one of the y-coordinates, ordered so that the Nth element of one list matches up with the Nth element of the other list. L1 and L2 are the default lists to use, and the List Editor (STAT > Edit...) is a useful window for entering the points.

The calculator does this regression by taking the natural log [ln\(](#) of the x-coordinates (this isn't stored anywhere) and then doing a linear regression. This means that if any x-coordinates are negative or 0, the calculator will instantly quit with [ERR:DOMAIN](#).

In its simplest form, LnReg takes no arguments, and fits a logarithmic curve through the points in L1 and L2:

... /

Menu Location

Press the LN key to paste $\ln($.

Calculator Compatibility

TI-83/84/+SE

Token Size

1 byte

LnReg

```
y=a+blnx  
a=-.0464694148  
b=6.412540931  
r2=.9131589664  
r=.9555935153
```

Command Summary

Calculates the best fit logarithmic curve through a set of points.

Command Syntax

```
LnReg [x-list, y-list, [frequency],  
[equation]]
```

Menu Location

```
: {9,13,21,30,31,31,34→L1  
: {260,320,420,530,560,550,590→L2  
: LnReg
```

On the home screen, or as the last line of a program, this will display the equation of the curve: you'll be shown the format, $y=a+b\ln(x)$, and the values of a and b. It will also be stored in the RegEQ variable, but you won't be able to use this variable in a program - accessing it just pastes the equation wherever your cursor was. Finally, the statistical variables a, b, r, and r^2 will be set as well. These latter two variables will be displayed only if "Diagnostic Mode" is turned on (see [DiagnosticOn](#) and [DiagnosticOff](#)).

You don't have to do the regression on L1 and L2, but if you don't you'll have to enter the names of the lists after the command. For example:

```
: {9,13,21,30,31,31,34→FAT  
: {260,320,420,530,560,550,590→CALS  
: LnReg ↵FAT,↵CALS
```

You can attach frequencies to points, for when a point occurs more than once, by supplying an additional argument - the frequency list. This list does not have to contain integer frequencies. If you add a frequency list, you must supply the names of the x-list and y-list as well, even when they're L1 and L2.

Finally, you can enter an equation variable (such as Y_1) after the command, so that the curve's equation is stored to this variable automatically. This doesn't require you to supply the names of the lists, but if you do, the equation variable must come last. You can use polar, parametric, or sequential variables as well, but since the equation will be in terms of X anyway, this doesn't make much sense.

An example of LnReg with all the optional arguments:

```
: {9,13,21,30,31,31,34→FAT  
: {260,320,420,530,560,550,590→CALS  
: {2,1,1,1,2,1,1→FREQ  
: LnReg ↵FAT,↵CALS,↵FREQ, Y1
```

Error Conditions

- [ERR:DOMAIN](#) is thrown if any x-coordinates are negative or 0.

Related Commands

- [ExpReg](#)
- [PwrReg](#)
- [SinReg](#)

Press:

1. STAT to access the statistics menu
2. LEFT to access the CALC submenu
3. 9 to select LnReg, or use arrows

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

The log(Command

The `log(` command computes the base 10 logarithm of a value — the exponent to which 10 must be raised, to get that value. This makes it the inverse of the `10^(` command.

`log(` is a real number for all positive real values. For negative numbers, `log(` is an imaginary number (so taking `log(` of a negative number will cause `ERR:NONREAL ANS` to be thrown in `Real` mode), and of course it's a complex number for complex values. `log(` is not defined at 0, even if you're in a complex mode.

Advanced Uses

Using either the `ln(` or the `log(` command, logarithms of any base can be calculated, using the identity:

$$\log_b x = \frac{\ln x}{\ln b} = \frac{\log_c x}{\log_c b} \quad (1)$$

So, to take the base B log of a number X, you could use either of the following equivalent ways:

```
:log(X)/log(B)
```

```
:ln(X)/ln(B)
```

This is the exponent to which B must be raised, to get X.

The base 10 logarithm specifically can be used to calculate the number of digits a whole number has:

```
:1+int(log(N))
```

This will return the number of digits N has, if N is a whole number. If N is a decimal, it will ignore the decimal digits of N.

Error Conditions

- `ERR:DOMAIN` when calculating `log(0)`.
- `ERR:NONREAL ANS` if taking `log(` of a negative number in `Real` mode.

<code>log(1)</code>	0
<code>log(10000)</code>	4
<code>log(2^5)/log(2)</code>	5

Command Summary

Computes the (principal branch of the) base 10 logarithm.

Command Syntax

`log(value)`

Menu Location

Press the LOG key to paste `log(`.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

Related Commands

- 10^(
- ln(

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/log>

The Logistic Command

Logistic tries to fit a logistic curve ($y=c/(1+ae^{-bx})$) through a set of points. To use it, you must first store the points to two lists: one of the x-coordinates and one of the y-coordinates, ordered so that the i th element of one list matches up with the i th element of the other list. L1 and L2 are the default lists used, and the List Editor (STAT > Edit...) is a useful window for entering the points.

The explanation for the odd format of a logistic curve is that it is the solution to a differential equation that models population growth with a limiting factor: a population that grows according to a logistic curve will start out growing exponentially, but will slow down before reaching a carrying capacity and approach this critical value without reaching it. The logistic curve also has applications, for example, in physics.

In its simplest form, Logistic takes no arguments, and fits a logistic curve through the points in L1 and L2:

```
:{9,13,21,30,31,31,34→L1  
:{260,320,420,530,560,550,590→L2  
:Logistic
```

On the home screen, or as the last line of a program, this will display the equation of the curve: you'll be shown the format, $y=c/(1+ae^{-bx})$, and the values of a , b and c . It will also be stored in the RegEQ variable, but you won't be able to use this variable in a program - accessing it just pastes the equation wherever your cursor was. Finally, the statistical variables a , b , and c will be set as well. There are no correlation statistics available for Logistic even if Diagnostic Mode is turned on (see DiagnosticOn and DiagnosticOff).

You do not have to do the regression on L1 and L2, in which case you will have to enter the names of the lists after the command. For example:

```
Logistic  
y=c/(1+ae^(-bx))  
a=14.50631042  
b=.6674283592  
c=16.01667428
```

Command Summary

Calculates the least-squares best fit logistic curve through a set of points.

Command Syntax

Logistic [*x-list*, *y-list*, [*frequency*],
[*equation*]]

Menu Location

Press:

1. STAT to access the statistics menu
2. LEFT to access the CALC submenu
3. ALPHA B to select Logistic, or use arrows

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

```
:{9,13,21,30,31,31,34→FAT
:{260,320,420,530,560,550,590→CALS
:Logistic ↵FAT,LCALS
```

You can attach frequencies to points, for when a point occurs more than once, by supplying an additional argument - the frequency list. This list does not have to contain integer frequencies. If you add a frequency list, you must supply the names of the x-list and y-list as well, even when they're L1 and L2.

Finally, you can enter an equation variable (such as Y_1) after the command, so that the curve's equation is stored to this variable automatically. This does not require you to supply the names of the lists, but if you do, the equation variable must come last. You can use polar, parametric, or sequential variables as well, but since the equation will be in terms of X anyway, this doesn't make much sense.

An example of Logistic with all the optional arguments:

```
:{9,13,21,30,31,31,34→FAT
:{260,320,420,530,560,550,590→CALS
:{2,1,1,1,2,1,1→FREQ
:Logistic ↵FAT,LCALS,LFREQ,Y1
```

Warning: if your data is not even slightly logistic in nature, then the calculator may return an error such as ERR:OVERFLOW. This happens when the calculator tries to calculate a carrying capacity, c , for the data, but since the rate of change in data doesn't seem to be slowing down, it assumes that the carrying capacity is still very far off, and tries large values for it. These values may get so large as to cause an overflow.

The [Levenberg-Marquardt](#) nonlinear least-squares algorithm is used by Logistic.

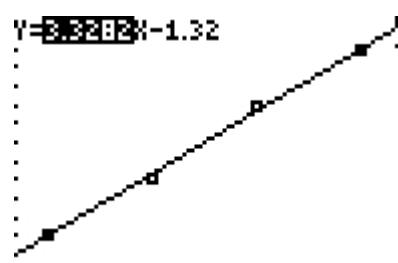
Related Commands

- [LinReg\(ax+b\)](#)
- [ExpReg](#)
- [SinReg](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/logistic>

The Manual-Fit Command

This command will allow the user to create a line of best fit according to their judgment. Activate the command by just pasting it on the screen. Then, click on a point for the line to begin followed by an end point. The calculator will then draw your line and display its equation at the top left corner of the screen. You can modify it by selecting the equation part and pressing enter. Input your desired value for the calculator to change it. The equation is stored into Y_1 . If you specify what equation you want to store it to, then it will store to that function.



Command Summary

```
:Manual-Fit  
(this activates the command and stor  
:Manual-Fit Y3  
(this stores to Y3 instead)
```

One note about this is that it only graphs linear models. It is written in the form $y=mx+b$, and you can modify m or b .
Exit out by 2nd QUIT.

Advanced Uses

This command is able to function in a program, but you cannot modify the values. This is a unique form of gathering user input that stores into the specified Y= function. Of course, this draws a line across the graph screen. You can then convert the function into a different form, like this:

```
:Manual-Fit  
:Equ►String(Y1,Str1)
```

This will turn the equation the user drew into a string which can then be used for output or calculations.

Related Commands

- [LinReg\(ax+b\)](#)
- [LinReg\(a+bx\)](#)
- [LinRegTInt](#)
- [LinRegTTest](#)
- [Med-Med](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/manual-fit>

The Matr►list(Command

The Matr►list(command stores one or more columns of a matrix (or expression resulting in a matrix) to list variables. The syntax is simple: first enter the matrix, then enter the list or lists you want to store columns to. The first (leftmost) column will be stored to the first list entered, the second column will be stored to the second list, and so on. For example:

Allows user to create a line of best fit and modify it

Command Syntax

Manual-Fit {Function}

Menu Location

Under Statistics or Catalog

1. Press STAT
2. Go to CALC
3. Press ALPHA D (or scroll to bottom)

Calculator Compatibility

TI-84+/SE (OS 2.30 or greater)

Token Size

2 bytes

```
Matr►list([[1,2]  
[3,4]],L1          Done  
L1                  {1 3}
```

Command Summary

```

[[11,12,13,14][21,22,23,24][31,32,33
  [[11 12 13 14]
   [21 22 23 24]
   [31 32 33 34]]
Matr►list(Ans,L1,L2
  Done
L1
  {11 21 31}
L2
  {12 22 32}

```

If there are more lists than columns in the matrix when doing Matr►list(), the extra lists will be ignored.

Matr►list() can also be used to extract a specific column of a matrix to a list. The order of the arguments is: matrix, column number, list name.

```

[[11,12,13,14][21,22,23,24][31,32,33
  [[11 12 13 14]
   [21 22 23 24]
   [31 32 33 34]]
Matr►list(Ans,4,L1
  Done
L1
  {14 24 34}

```

Advanced Uses

While the command deals with columns, occasionally you might want to store the matrix to lists by rows. The T (transpose) command is your friend here: applying it to the matrix will flip it diagonally, so that all rows will become columns and vice-versa. For example:

```

[[11,12,13,14][21,22,23,24][31,32,33,34
  [[11 12 13 14]
   [21 22 23 24]
   [31 32 33 34]]
Matr►list(AnsT,L1,L2
  Done
L1
  {11 12 13 14}
L2
  {21 22 23 24}

```

Optimizations

When using Matr►list() to store to named lists, only the first list must have an L in front of its name — it can be omitted for the rest. For example:

Command Summary

Stores one or more columns of a given matrix to list variables

Command Syntax

`Matr►list(matrix, list-var1, [list-var2, ...])`
`Matr►list(matrix, column#, list-var)`

Menu Location

Press:

1. MATRIX (on the 83) or 2nd MATRIX (83+ or higher) to access the matrix menu
2. LEFT to access the MATH submenu
3. 8 to select Matr►list(), or use arrows.

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

```
:Matr►list([A],COL1,COL2,COL3  
can be  
:Matr►list([A],COL1,COL2,COL3
```

On the other hand, when storing a specific column of a matrix to a named list, the list does not need to be preceded by an `L`.

```
:Matr►list([A],N,COL1  
can be  
:Matr►list([A],N,COL1
```

Related Commands

- [List►matr\(](#)
- [T \(transpose\)](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/matrlist>

The max(Command

`max(x,y)` returns the largest of the two numbers `x` and `y`. `max(list)` returns the largest element of `list`. `max(list1,list2)` returns the pairwise maxima of the two lists. `max(list1,x)` (equivalently, `max(x,list1)`) returns a list whose elements are the larger of `x` or the corresponding element of the original list.

```
max(2,3)  
3  
max({2,3,4})  
4  
max({1,3},{4,2})  
{4 3}  
max({1,3},2)  
{2 3}
```

Unlike comparison operators such as `<` and `>`, `max` can also compare complex numbers. To do this, both arguments must be complex — either complex numbers or complex lists: `max(2,i)` will throw an error even though `max(2+0i,i)` won't. In the case of complex numbers, the number with the largest absolute value will be returned. When the two numbers have the same absolute value, the first one will be returned: `max(i,-i)` returns `i` and `max(-i,i)` returns `-i`.

Advanced Uses

```
max(2,9  
9  
max({2,4,4,3  
4  
max(5,{1,6,9  
5 6 9)
```

Command Summary

Returns the maximum of two elements or of a list.

Command Syntax

- for two numbers: `max(x,y)`
- for a list: `max(list)`
- comparing a number to each element of a list: `max(x,list)` or `max(list,x)`
- pairwise comparing two lists: `max(list1,list2)`

Menu Location

Press:

1. MATH to access the [math](#) menu.
2. RIGHT to access the NUM

`max(` can be used in Boolean comparisons to see if at least one of a list is 1 (true) — useful because commands like If or While only deal with numbers, and not lists, but comparisons like `L1=L2` return a list of values. In general, the behavior you want varies, and you will use the `min(` function or the `max(` function accordingly.

Using `max(` will give you a lenient test — if any one element of the list is 1 (true), then the `max(` of the list is true — this is equivalent to putting an 'or' in between every element. For example, this tests if `K` is equal to any of 24, 25, 26, or 34 (the getKey arrow key values):

```
: If max(K={24,25,26,34
: Disp "ARROW KEY
```

- submenu.
3. 7 to select `max(`, or use arrows.

Alternatively, press:

1. 2nd LIST to access the list menu.
2. LEFT to access the MATH submenu.
3. 2 to select `max(`, or use arrows.

Calculator Compatibility

TI-83/84/+SE

Token Size

1 byte

Error Conditions

- **ERR:DATA TYPE** is thrown when comparing a real and a complex number. This can be avoided by adding `+0i` to the real number (or `i^4` right after it, for those who are familiar with complex numbers)
- **ERR:DIM MISMATCH** is thrown, when using `max(` with two lists, if they have different dimensions.

Related Commands

- `min(`
- `sum(`
- `prod(`

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/max>

The `mean(` Command

The `mean(` command finds the mean, or the average, of a list. It's pretty elementary. It takes a list of real numbers as a parameter. For example:

```
: Prompt L1
: Disp "MEAN OF L1",mean(L1
```

That's not all, however. Awesome as the `mean(` command is, it can also take a frequency list argument, for situations when your elements occur more than once. For example:

```
mean({1,2,3,10      4
mean(L1            2
```

Command Summary

Finds the mean (the average) of a list.

Command Syntax

```
:Disp mean({1,2,3},{5,4,4})
```

is short for

```
:mean({1,1,1,1,1,2,2,2,2,3,3,3,3})
```

The frequency list {5,4,4} means that the first element, 1, occurs 5 times, the second element, 2, occurs 4 times, and the third element, 3, occurs 4 times.

Advanced Uses

You can also use the frequency list version of mean() to calculate weighted averages. For example, suppose you're trying to average grades in a class where homework is worth 50%, quizzes 20%, and tests 30%. You have a 90% average on homework, 75% on quizzes (didn't study too well), but 95% average on tests. You can now calculate your grade with the mean() command:

```
:mean({90,75,95},{50,20,30})
```

You should get a 88.5 if you did everything right.

Frequency lists don't need to be whole numbers. Amazing as that may sound, your calculator can handle being told that one element of the list occurs 1/3 of a time, and another occurs 22.7 times. It can even handle a frequency of 0 - it will just ignore that element, as though it weren't there. In particular, mean(L1,L2) is effectively equivalent to sum(L1*L2)/sum(L2).

One caveat, though - if all of the elements occur 0 times, there's nothing to take an average of and your calculator will throw an error.

Error Conditions

- **ERR:DATA TYPE** is thrown, among other cases, if the data list is complex, or if the frequencies are not all positive and real.
- **ERR:DIM MISMATCH** is thrown if the frequency list and the data list have a different number of elements.
- **ERR:DIVIDE BY 0** is thrown if the frequency list's elements are all 0.

Related Commands

- [median\(\)](#)
- [stdDev\(\)](#)
- [variance\(\)](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/mean>

Command Syntax

`mean(list,[freqlist])`

Menu Location

Press:

1. 2ND LIST to enter the LIST menu.
2. LEFT to enter the MATH submenu.
3. 3 to select mean(), or use arrows.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

The median(Command

The median(command finds the median of a list. It takes a list of real numbers as a parameter. For example:

```
:Prompt L1  
:Disp "MEDIAN OF L1",median(L1)
```

That's not all, however. Awesome as the median(command is, it can also take a frequency list argument, for situations when your elements occur more than once. For example:

```
:Disp median({1,2,3},{5,4,4})
```

is short for

```
:median({1,1,1,1,1,2,2,2,2,3,3,3})
```

The frequency list {5,4,4} means that the first element, 1, occurs 5 times, the second element, 2, occurs 4 times, and the third element, 3, occurs 4 times.

Advanced Uses

Frequency lists don't need to be whole numbers. Amazing as that may sound, your calculator can handle being told that one element of the list occurs 1/3 of a time, and another occurs 22.7 times. It can even handle a frequency of 0 - it will just ignore that element, as though it weren't there. One caveat, though - if all of the elements occur 0 times, there's nothing to take an average of and your calculator will throw an error.

Error Conditions

- **ERR:DATA TYPE** is thrown, among other cases, if the data list is complex, or if the frequencies are not all positive and real.
- **ERR:DIM MISMATCH** is thrown if the frequency list and the data list have a different number of elements.
- **ERR:DIVIDE BY 0** is thrown if the frequency list's elements are all 0.

Related Commands

- [mean\(](#)
- [stdDev\(](#)

```
median({1,2,3,10}  
2.5  
median(L1)  
2
```

Command Summary

Finds the median of a list.

Command Syntax

median(*list*,[*freqlist*])

Menu Location

Press:

1. 2ND LIST to enter the LIST menu.
2. LEFT to enter the MATH submenu.
3. 4 to select median(, or use arrows.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

- variance(

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/median>

The Med-Med Command

The Med-Med command is one of several that can calculate a line of best fit through a set of points. However, unlike the LinReg(ax+b) and LinReg(a+bx) commands, which generate the same result in different formats, Med-Med produces a different line entirely, known as the 'median fit line' or the 'median-median model'. This model is more resistant to outliers than the best-fit line produced by LinReg(ax+b)-type commands, in much the same way that the median of a set of data is more resistant to outliers than the mean. The process of calculating a median fit line is roughly as follows:

1. Divide the data into three equal groups by their x-values (the smallest third, the middle third, and the largest third)
2. Find the "median point" for each group by pairing the median x-value in the group with the median y-value (this need not be an actual data point).
3. These points are stored to (x_1, y_1) , (x_2, y_2) , and (x_3, y_3) on the calculator.
4. Find the line passing through the median point for the first and third group.
5. Shift this line one-third of the way toward the median point of the second group.

To use the Med-Med command, you must first store the points to two lists: one of the x-coordinates and one of the y-coordinates, ordered so that the Nth element of one list matches up with the Nth element of the other list. L1 and L2 are the default lists to use, and the List Editor (STAT > Edit...) is a useful window for entering the points. As you can see from the steps shown above, Med-Med requires at least three points with different x-values to work with.

In its simplest form, Med-Med takes no arguments, and calculates a regression line through the points in L1 and L2:

```
: {9,13,21,30,31,31,34→L1
: {260,320,420,530,560,550,590→L2
: Med-Med
```

Med-Med
 $y=ax+b$
 $a=2.096051101$
 $b=-1.535803195$

Command Summary

Calculates the median fit line through a set of points.

Command Syntax

Med-Med [*x-list*, *y-list*, [*frequency*], [*equation*]]

Menu Location

Press:

1. STAT to access the statistics menu
2. LEFT to access the CALC submenu
3. 3 to select Med-Med, or use arrows

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

On the home screen, or as the last line of a program, this will display the equation of the regression line: you'll be shown the format, $y=ax+b$, and the values of a and b . It will also be stored in the RegEQ variable, but you won't be able to use this variable in a program -

accessing it just pastes the equation wherever your cursor was. Finally, the statistical variables a and b will be set as well. There are no diagnostics available for the Med-Med command, so r and r^2 will not be calculated or displayed even if you run [DiagnosticOn](#).

You don't have to do the regression on L1 and L2, but if you don't you'll have to enter the names of the lists after the command. For example:

```
: {9,13,21,30,31,31,34→FAT  
: {260,320,420,530,560,550,590→CALS  
: Med-Med ↵FAT,↵CALS
```

You can attach frequencies to points, for when a point occurs more than once, by supplying an additional argument - the frequency list. This list does not have to contain integer frequencies. If you add a frequency list, you must supply the names of the x-list and y-list as well, even when they're L1 and L2.

Finally, you can enter an equation variable (such as Y_1) after the command, so that the line of best fit is stored to this equation automatically. This doesn't require you to supply the names of the lists, but if you do, the equation variable must come last. You can use polar, parametric, or sequential variables as well, but since the line of best fit will be in terms of X anyway, this doesn't make much sense.

An example of Med-Med with all the optional arguments:

```
: {9,13,21,30,31,31,34→FAT  
: {260,320,420,530,560,550,590→CALS  
: {2,1,1,1,2,1,1→FREQ  
: Med-Med ↵FAT,↵CALS,↵FREQ, Y1
```

Related Commands

- [LinReg\(ax+b\)](#)
- [LinReg\(a+bx\)](#)
- [LinRegTTest](#)
- [LinRegTInt](#)
- [Manual-Fit](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/med-med>

The Menu(Command

Menus are used for organization, to provide a list of choices for the user to select from, as well as a good way for users to interact with and navigate programs. Although using the `Menu(` command requires branching (which is generally frowned upon in most circumstances), the menu looks like a generic built-in menu, so it is familiar and easy to use for the user.

PROGRAM: EXAMPLE
`:Menu("ARE YOU SURE?", "YES", 1, "NO", 2
:Lbl 1
:Lbl 2`

When the `Menu(` command is encountered during a program, the menu screen is displayed with the specified menu title in white-on-black text on the top line and each menu item listed below on its own line, the pause indicator turns on, and execution pauses until the user selects a menu item. There is a cursor that the user can move up and down the menu to select a menu item.

The menu title can be sixteen characters or less (because of the screen width), and must be enclosed in a pair of quotation marks. The menu title looks best if you center it on the screen (using spaces to fill in the rest of the line), so that the entire top line will be highlighted. The menu can have up to seven menu items (because of the screen height and the menu title on top).

After the menu title, you put a comma and then the menu items. There are two parts to a menu item: the text that will be displayed on the screen and the label that program execution will continue at if the user presses `ENTER` on the menu item or presses its corresponding number. The text can be fourteen characters or less (because the menu item number is displayed on the left) and must be enclosed in a pair of quotation marks, and you have to separate the text and label with a comma.

```
PROGRAM:MENU
:Lbl NY
:Menu(" Select A Place ", "NY", NY, "LA", NY, "MN", MN
:Lbl MN
:Disp "Good Choice!"
```

Advanced Uses

When a program needs more than seven menu items, you will have to create another menu and then link to that menu from the first menu with one of the menu items. Similarly, you can also have two menu items go to the same label (you do not need two labels if they are right next to each other).

If you get tired of using the `Menu(` command every time you want to make a menu, the alternative is to make your own custom menu. A custom menu provides a richer experience for the user, and isn't much more work than using the `Menu(` command. In addition, as you get more experienced as a programmer, you'll come to enjoy using custom menus.

You can use a string variable for the menu title and menu item text instead of the text in quotes, which may sometimes be smaller if the text is used at other places in the program. Similarly, its possible to place all the menu titles in one string variable, and then just access the respective menu title as a substring. Unfortunately, variables will not work for the menu item labels. You can also leave the menu title blank to give the illusion that there is no menu title by using two quotes side by side (i.e. "").

For many programs, including text-based programs (where menus are heavily used), there is a main menu that is used for navigating to the different parts of the program. While each

Command Summary

Displays a generic menu on the home screen, with up to seven options for the user to select.

Command Syntax

```
Menu("Title", "Option 1", Label 1[,  
..., "Option 7", Label 7])
```

Menu Location

While editing a program, press:

1. `PRGM` to enter the `PRGM` menu
2. Press `ALPHA PRGM` to choose `Menu(` or use arrows

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

program's main menu is unique, two of the most standard menu items on a main menu are Start and Quit — Start goes to the beginning of the program, while Quit goes to the end. It is also fairly common to place a label right before the main menu, so you can return to it again later in the program.

Problems

1. There is only one line for the title.
 2. There are only seven slots for the menu items and no scrolling (you CAN add a "next" at the bottom, but that just looks bad, especially if you have a "back" and/or "exit" down there already)
 3. The screen refreshes or blinks when you press down at the bottom to go back to the top.
 4. During the loading of the menu, you can see what is written on the home screen.

Optimization

Because the `Menu(` command displays the menu screen instead of clearing the home screen, you do not need to put the `ClrHome` command before it.

```
:ClrHome  
:Menu("Choose", "Right", 1, "Wrong", 2  
Remove ClrHome  
:Menu("Choose", "Right", 1, "Wrong", 2
```

Error Conditions

- **ERR:INVALID** occurs if this statement is used outside a program.
 - **ERR:LABEL** is thrown when an option is chosen whose label doesn't exist.

Related Commands

- Goto/Lbl

See Also

- Custom Menus

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/menu>

The min(Command

`min(x,y)` returns the smallest of the two numbers `x` and `y`. `min(list)` returns the smallest element of `list`. `min(list1,list2)` returns the pairwise minima of the two lists. `min(list1,x)` (equivalently, `min(x,list1)`) returns a list whose elements are the smaller of `x` or the corresponding element of the original list.

`min(2,3)`

```
min(2,9)      2  
min(1,2,3,8) 1  
min(0,-5,6,-2)  
              -5 0 -2
```

```

2
min({2,3,4})
2
min({1,3},{4,2})
{1 2}
min({1,3},2)
{1 2}

```

Unlike relational operators, such as $<$ and $>$, `min()` can also compare complex numbers. To do this, both arguments must be complex — either complex numbers or complex lists: `min(2,i)` will throw a ERR:DATA TYPE error even though `min(2+0i,i)` won't. In the case of complex numbers, the number with the smallest absolute value will be returned. When the two numbers have the same absolute value, the second one will be returned: `min(i,-i)` returns $-i$ and `min(-i,i)` returns i .

Advanced Uses

`min()` can be used in Boolean comparisons to see if every value of a list is 1 (true) — useful because commands like If or While only deal with numbers, and not lists, but comparisons like `L1=L2` return a list of values. In general, the behavior you want varies, and you will use the `min()` or max() functions accordingly.

Using `min()` will give you a strict test — only if every single value of a list is true will `min()` return true. For example, the following code will test if two lists are identical — they have the same exact elements — and print EQUAL in that case:

```

: If dim(L1)=dim(L2
: Then
: If min(L1=L2
: Disp "EQUAL
: End

```

The first check, to see if the sizes are identical, is necessary because otherwise comparing the lists will return a ERR:DIM MISMATCH error.

Error Conditions

- ERR:DATA TYPE is thrown when comparing a real and a complex number. This can be avoided by adding $0i$ to the real number.
- ERR:DIM MISMATCH is thrown, when using `min()` with two lists, if they have different dimensions.

Related Commands

Command Summary

Returns the minimum of two elements or of a list.

Command Syntax

- for two numbers: `min(x,y)`
- for a list: `min(list)`
- comparing a number to each element of a list: `min(x,list)` or `min(list,x)`
- pairwise comparing two lists: `min(list1,list2)`

Menu Location

Press:

1. MATH to access the math menu.
2. RIGHT to access the NUM submenu.
3. 6 to select `min()`, or use arrows.

Alternatively, press:

1. 2nd LIST to access the list menu.
2. LEFT to access the MATH submenu.
3. ENTER to select `min()`.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

- max(
- sum(
- prod(

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/min>

The nCr Command

nCr is the number of combinations function (or binomial coefficient), defined as $a \text{ nCr } b = a!/(b!(a-b)!)$, where a and b are nonnegative integers. The function also works on lists.

Tip: nCr has a higher priority in evaluation than operators such as + or *: for example, $5X \text{ nCr } 10$ will be interpreted as $5(X \text{ nCr } 10)$ and not as $(5X) \text{ nCr } 10$. You might wish to use parentheses around complex expressions that you will give to nCr as arguments.

```
6 nCr 4
      15
```

The combinatorial interpretation of $a \text{ nCr } b$ is the number of ways to choose b objects, out of a total of a , if order doesn't matter. For example, if there are 10 possible pizza toppings, there are $10 \text{ nCr } 3$ ways to choose a 3-topping pizza.

Error Conditions

- **ERR:DIM MISMATCH** is thrown when applying nCr to two lists that have different dimensions.
- **ERR:DOMAIN** is thrown for negative integers or decimals.

Related Commands

- nPr
- !

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/ncr>

5 nCr 3	10
5 nCr 2	10
4 nCr {0,1,2,3,4}	{1 4 6 4 1}
3	

Command Summary

Calculates the combinatorial number of combinations.

Command Syntax

$a \text{ nCr } b$

Menu Location

Press:

1. MATH to access the math menu.
2. LEFT to access the PRB submenu.
3. 3 to select nCr, or use arrows.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

The nDeriv(Command

nDeriv($f(var), var, value[, h]$) computes an approximation to the value of the derivative of $f(var)$

```
nDeriv(e^(Z), Z, 1)
```

with respect to *var* at *var=value*. *h* is the step size used in the approximation of the derivative. The default value of *h* is 0.001.

`nDeriv` only works for real numbers and expressions. `nDeriv` can be used only once inside another instance of `nDeriv`.

Tip: *h* should not be set too small or roundoff error can affect your result. For a suggestion on how to choose *h* for acceptable accuracy, see [this post](#) on UTI.

```
π→X  
3.141592654  
nDeriv(sin(T),T,X)  
-.9999998333  
nDeriv(sin(T),T,X,(abs(X)+e-6)e-6)  
-1.000000015  
nDeriv(nDeriv(cos(U),U,T),T,X)  
.999999665
```

Formulas

The exact formula that the calculator uses to evaluate this function is:

$$\text{nDeriv}(f(t),t,x,h) = \frac{f(x+h) - f(x-h)}{2h} \quad (1)$$

(.001 is substituted for *h* when the argument is omitted)

Error Conditions

- **ERR:DOMAIN** is thrown if *h* is 0 (since this would yield division by 0 in the formula)
- **ERR:ILLEGAL NEST** is thrown if `nDeriv` commands are nested more than one level deep. Just having one `nDeriv` command inside another is okay, though.

Related Commands

- [`fMin`](#)
- [`fMax`](#)
- [`fnInt`](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/nderiv>

```
2.718282282  
nDeriv(e^(Z),Z,1  
,.05 2.719414587
```

Command Summary

Calculates the approximate numerical derivative of a function, at a point.

Command Syntax

`nDeriv(f(variable),variable,value[,h])`

Menu Location

Press:

1. MATH to access the [math](#) menu.
2. 8 to select `nDeriv`, or use arrows.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

The ► Nom(Command

The ►Nom(command converts from an effective interest rate to a nominal interest rate. In other words, it converts an interest rate that takes compounding periods into account into one that doesn't. The two arguments are 1) the interest rate and 2) the number of compounding periods.

For example, you want to know the interest rate, compounded monthly, that will yield a total increase of 10% per year:

```
►Nom(10,12)  
9.568968515
```

Formulas

The formula for converting from an effective rate to a nominal rate is:

$$\text{Nom} = 100 \text{ CP} \left(\sqrt[\text{CP}]{\frac{\text{Eff}}{100} + 1} - 1 \right) \quad (1)$$

Here, Eff is the effective rate, Nom is the nominal rate, and CP is the number of compounding periods.

Error Conditions

- **ERR:DOMAIN** is thrown if the number of compounding periods is not positive, or if the nominal rate is -100% or lower (an exception's made for the nominal rate if there is only one compounding period, since ►Nom(X,1)=X).

Related Commands

- ►Eff(

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/nom>

```
►Nom(200,12  
115.0472294  
►Nom(100,12  
71.35571323  
►Eff(Ans,12  
100
```

Command Summary

Converts an effective interest rate to a nominal interest rate.

Command Syntax

►Nom(*interest rate,compounding periods*)

Menu Location

On the TI-83, press:

1. 2nd FINANCE to access the finance menu.
2. ALPHA B to select ►Nom(.

On the TI-83+ or higher, press:

1. APPS to access the applications menu.
2. ENTER or 1 to select Finance...
3. ALPHA B to select ►Nom(.

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

The normalcdf(Command

normalcdf(is the normal (Gaussian) cumulative

density function. If some random variable follows a normal distribution, you can use this command to find the probability that this variable will fall in the interval you supply.

There are two ways to use normalcdf(. With two arguments (lower bound and upper bound), the calculator will assume you mean the standard normal distribution, and use that to find the probability corresponding to the interval between "lower bound" and "upper bound". You can also supply two additional arguments to use the normal distribution with a specified mean and standard deviation. For example:

```
for the standard normal distribution  
:normalcdf(-1,1  
  
for the normal distribution with mean  
:normalcdf(5,15,10,2.5
```

Advanced

Often, you want to find a "tail probability" - a special case for which the interval has no lower or no upper bound. For example, "what is the probability x is greater than 2?". The TI-83+ has no special symbol for infinity, but you can use E99 to get a very large number that will work equally well in this case (E is the decimal exponent obtained by pressing [2nd] [EE]). Use E99 for positive infinity, and -E99 for negative infinity.

The normal distribution is often used to approximate the binomial distribution when there are a lot of trials. This isn't really necessary on the TI-83+ because the binompdf(and binomcdf(commands are already very fast - however, the normal distribution can be slightly faster, and the skill can come in handy if you don't have access to a calculator but do have a table of normal distributions (yeah, right). Here is how to convert a binomial distribution to a normal one:

```
:binompdf(N,P,X  
can be  
:normalcdf(X-.5,X+.5,NP,√(NP(1-P  
  
:binomcdf(N,P,X,Y  
can be  
:normalcdf(X-.5,Y+.5,NP,√(NP(1-P
```

How much faster this is will depend on N and P, since the binomial distribution takes a long time to evaluate for large values of N, but the normal distribution takes about the same time for any mean and standard deviation. Also, this is an approximation that is only valid for some binomial distributions - a common rule of thumb is NP>10.

```
normalcdf(1,e99  
.1586552596  
normalcdf(30,35,  
33,2  
.7745375117
```

Command Summary

Finds the probability for an interval of the normal curve.

Command Syntax

`normalpdf(lower, upper [, μ, σ])`

Menu Location

Press:

1. 2ND DISTR to access the distribution menu
2. 2 to select normalcdf(, or use arrows.

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

Formulas

As with other continuous distributions, any probability is an integral of the probability density function. Here, too, we can define `normalcdf(` for the standard normal case in terms of `normalpdf(`:

$$\text{normalcdf}(a, b) = \int_a^b \text{normalpdf}(x) dx = \frac{1}{\sqrt{2\pi}} \int_a^b e^{-\frac{1}{2}x^2} dx \quad (1)$$

or in terms of the error function:

$$\text{normalcdf}(a, b) = \frac{1}{2} \left(\text{erf}\left(\frac{b}{\sqrt{2}}\right) - \text{erf}\left(\frac{a}{\sqrt{2}}\right) \right) \quad (2)$$

For the arbitrary mean μ and standard deviation σ , `normalcdf(` is defined in terms of the standard normal distribution, with the bounds of the interval standardized:

$$\text{normalcdf}(a, b, \mu, \sigma) = \text{normalcdf}\left(\frac{a - \mu}{\sigma}, \frac{b - \mu}{\sigma}\right) \quad (3)$$

Related Commands

- [normalpdf\(](#)
- [invNorm\(](#)
- [ShadeNorm\(](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/normalcdf>

The Normal Command

The `Normal` command puts the calculator in normal number mode, in which it only uses scientific notation for large enough numbers (10 000 000 000 or higher), negative numbers large enough in absolute value (-10 000 000 000 or lower), or numbers close enough to 0 (less than .001 and greater than -.001)

The other possible settings are [Sci](#) (which always uses scientific notation), or [Eng](#) (which uses a specific form of scientific notation based on powers of 1000)

Related Commands

- [Sci](#)
- [Eng](#)
- [Float](#)

10000	Done
	1e4
Normal	Done
10000	Done
	10000

Command Summary

Puts the calculator in "normal" mode regarding scientific notation.

Command Syntax

`Normal`

Menu Location

- Fix

While editing a program, press:

1. MODE to access the mode menu.
2. ENTER to select Normal.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/normal>

The normalpdf(Command

normalpdf(is the normal (Gaussian) probability density function.

Since the normal distribution is continuous, the value of normalpdf(doesn't represent an actual probability - in fact, one of the only uses for this command is to draw a graph of the normal curve. You could also use it for various calculus purposes, such as finding inflection points.

The command can be used in two ways:
normalpdf(x) will evaluate the standard normal p.d.f. (with mean at 0 and a standard deviation of 1) at x, and normalpdf(x, μ , σ) will work for an arbitrary normal curve, with mean μ and standard deviation σ .

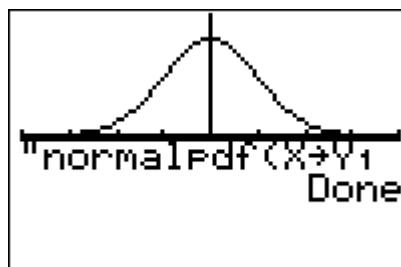
Formulas

For the standard normal distribution, normalpdf(x) is defined as

$$\text{normalpdf}(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2} \quad (1)$$

For other normal distributions, normalpdf(is defined in terms of the standard distribution:

$$\text{normalpdf}(x, \mu, \sigma) = \frac{1}{\sigma} \text{normalpdf}\left(\frac{x - \mu}{\sigma}\right) \quad (2)$$



Command Summary

Evaluates the normal probability density function at a point.

Command Syntax

normalpdf(x[, μ , σ])

Menu Location

Press:

1. 2ND DISTR to access the distribution menu
2. ENTER to select normalpdf(.

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

Related Commands

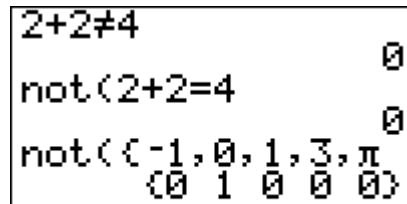
- [normalcdf\(](#)
- [invNorm\(](#)
- [ShadeNorm\(](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/normalpdf>

The not(Command

The last logical operator available on the 83 series takes only one value as input. **not(** comes with its own parentheses to make up for this loss. Quite simply, it negates the input: False becomes True (1) and True returns False (0). **not(** can be nested; one use is to make any True value into a 1.

```
:not(0)  
1  
  
:not(-20 and 14)  
0  
  
:not(not(2))  
1
```



```
2+2#4  
0  
not(2+2=4  
0  
not((-1,0,1,3,π))  
(0 1 0 0 0)
```

Command Summary

Flips the truth value of its argument.

Command Syntax

`not(value)`

Menu Location

Press:

1. 2nd TEST to access the test menu.
2. RIGHT to access the LOGIC submenu.
3. 4 to select not(, or use arrows and ENTER.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

Advanced Uses

`not(not(X))` will make any value X into 1 if it's not 0, and will keep it 0 if it is.

Optimization

`not(X)` and `X=0` have the same truth value, but `not(` is shorter if the closing parenthesis is omitted:

```
:If A=0  
can be  
:If not(A
```

Related Commands

- [and](#)
- [or](#)
- [xor](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/not>

The nPr Command

nPr is the number of permutations function, defined as $a \text{ nPr } b = a!/(a-b)!$, where a and b are nonnegative integers. The function also works on lists.

Tip: nPr has a higher priority in evaluation than operators such as + or *: for example, $5X \text{ nPr } 10$ will be interpreted as $5(X \text{ nPr } 10)$ and not as $(5X) \text{ nPr } 10$. You might wish to use parentheses around complex expressions that you will give to nPr as arguments.

```
6 nPr 4  
360
```

The combinatorial interpretation of $a \text{ nPr } b$ is the number of ways to choose b objects in order, when there are a objects in total. For example, when giving 1st, 2nd, and 3rd place awards in a competition between 10 teams, there are $10 \text{ nPr } 3$ different ways to assign the awards.

Error Conditions

- **ERR:DIM MISMATCH** is thrown when applying nPr to two lists that have different dimensions.
- **ERR:DOMAIN** is thrown for negative integers or decimals.

Related Commands

- [nCr](#)
- [!](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/npr>

5 nPr 3	60
5 nPr 2	20
4 nPr {0,1,2,3,4}	
{1 4 12 24 24}	

Command Summary

Calculates the combinatorial number of permutations.

Command Syntax

$a \text{ nPr } b$

Menu Location

Press:

1. MATH to access the [math](#) menu.
2. LEFT to access the PRB submenu.
3. 2 to select nPr, or use arrows.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

The npv(Command

The npv(command computes the net present value

of money over a specified time period. If a positive value is returned after executing `npv()`, that means it was a positive cashflow; otherwise it was a negative cashflow. The `npv()` command takes four arguments, and the fourth one is optional:

- **interest rate** — the percentage of the money that is paid for the use of that money over each individual period of time.
- **CF0** — the initial amount of money that you start out with; this number must be a real number, otherwise you will get a ERR:DATA TYPE error.
- **CFLIST** — the list of cash flows added or subtracted after the initial money.
- **CFFreq** — the list of frequencies of each cash flow added after the initial money; if this is left off, each cash flow in the cash flow list will just appear once by default.

Sample Problem

Your mom recently opened a bank account for you, with 0 as a gift to start you off. This is welcome news to you, until you find out that the bank charges 5% as the interest rate for the account. So, you get a job at Rocco's Pizzas delivering pizzas, which brings in \$1250, \$1333, \$1575, \$1100, and \$1900 each month. For the last five months, in particular, you have earned \$1250, \$1333, \$1575, \$1100, and \$1900. (Assume there are no other expenses, such as gas, car payment, etc.)

Plugging in all of the different values into the `npv()` command, this is what our code looks like:

```
:npv(5,500,{1250,1333,1575,1100,1900}
```

```
npv(10,5E3,E3C5,-2,1,-4,3},{1,2,3,4,5
5485.652667
npv(10,5E3,E3C5,-2,1,-4,3
7774.586932
```

Command Summary

Computes the net present value of money over a specified time period.

Command Syntax

```
npv(interest  
rate,CF0,CFLIST[,CFFreq])
```

Menu Location

Press:

1. 2nd FINANCE (on a TI-83) or APPS (TI-83+ or higher) to access the finance menu
2. On the TI-83+ or higher, select the first option "Finance..." from the APPS menu
3. 7 or use arrow keys to scroll to it

Calculator Compatibility

TI-83/84/+SE

Token Size

2 bytes

Optimization

The `npv()` command's optional fourth argument should be left off if each cash flow of money in the list of cash flows just appears once.

```
:npv(5,1550,{2E3,3E3,4E3},{1,1,1
can be
:npv(5,1550,{2E3,3E3,4E3
```

At the same time, if there are cash flows that occur multiple times, it can be smaller to just use the frequency argument:

```
:npv(8,0,{200,200,300,300,300
```

can be
`:npv(8,0,{200,300},{2,3}`

Formulas

Without a frequency list, the formula for npv(is the following:

$$\text{npv}(i, \text{CF}_0, \{\text{CF}_j\}) = \sum_{j=0}^N \text{CF}_j \left(1 + \frac{i}{100}\right)^{-j} \quad (1)$$

When a frequency list is used, the same formula can be applied if we expand the list with frequencies into a long list without frequencies. However, it's possible to do the calculation directly. We define the cumulative frequency S_j as the sum of the first j frequencies (S_0 is taken to be 0):

$$\text{npv}(i, \text{CF}_0, \{\text{CF}_j\}, \{n_j\}) = \text{CF}_0 + \sum_{j=1}^N \text{CF}_j \left(1 + \frac{i}{100}\right)^{S_{j-1}} \frac{(1 - (1 + \frac{i}{100})^{-n_j})}{i} \quad (2)$$

Error Conditions

- **ERR:DATA TYPE** is thrown if you try to use anything other than a real number for the interest rate.
- **ERR:DIM MISMATCH** is thrown if the list of cash flows and the list of cash flow frequencies have different dimensions.

Related Commands

- [irr\(\)](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/npv>

The OpenLib(Command

Together with [ExecLib](#), OpenLib(is used on the TI-84 Plus and TI-84 Plus SE for running routines from a Flash App library. This only works, of course, with libraries that have been specifically written for this purpose. The only such library so far is [usb8x](#), for advanced interfacing with the USB port.

The following program, which displays the version of usb8x, is an example of how to use OpenLib(and ExecLib:

```
:OpenLib(USBDRV8X
:{6
```

```
PROGRAM:OPENLIB
:OpenLib(USBDRV8
X
:{6
:ExecLib
:Ans(2)+.01Ans(3
```

Command Summary

Sets up a compatible Flash application library for use with ExecLib

```
:ExecLib  
:Ans(2)+.01Ans(3)
```

Related Commands

- [ExecLib](#)

Command Syntax

OpenLib(*library*)

Menu Location

This command is only found in the catalog menu. Press:

1. 2nd CATALOG to access the command catalog.
2. O to skip to commands starting with O.
3. ENTER to select OpenLib(.)

Calculator Compatibility

TI-84+/SE

Token Size

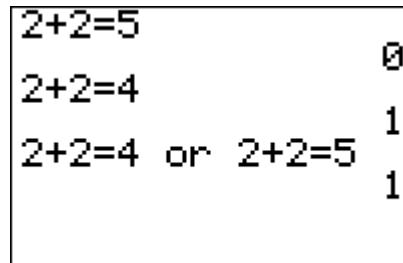
2 bytes

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/openlib>

The or Command

or takes two numbers or expressions, and checks to see if *either one* is True. If both are False, it returns 0. If at least one is True, it returns 1. **or** is commutative (i.e. the order of arguments does not matter). As with [and](#), you can use variables and expressions, and use multiple **or**'s together.

```
:0 or 1-1  
0  
:0 or -1  
1  
:2 or 6*4  
1  
:0 or 1 or 0  
1
```



2+2=5
0
2+2=4
1
2+2=4 or 2+2=5
1

Command Summary

Returns the truth value of *value1* or *value2* being true.

Command Syntax

value1 or *value2*

Menu Location

Press:

1. 2nd TEST to access the test menu.
2. RIGHT to access the LOGIC submenu.

Related Commands

- [and](#)
- [xor](#)

- not(

3. 2 to select or, or use arrows and ENTER.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/or>

The Output(Command

The Output(command is the fastest way to display text on the home screen. It takes three arguments: the row (1-8) at which you want to display something, the column (1-16), and whatever it is you want to display. It allows for more freedom than the Disp command.

Although values offscreen for the row and column values will cause an error, it's okay if part of the text displayed goes off the screen. When text goes past the last (16th) column, it will wrap to the first column of the next row. If the text goes past the last column of the last row, the remainder will be truncated. Output(will never cause the screen to scroll.

When the horizontal screen split mode is activated, only the first four rows of the home screen are available for the Output(command, which may cause undesirable behavior, and trying to output to the last four rows will cause an error. It is advisable to use the Full command at the beginning of a program that relies on Output(.

Like other text display commands, you can display each function and command as text. However, this is not without problems as each function and command is counted as one character. The two characters that you can't display are quotation marks ("") and the store command (→). However, you can mimic these respectively by using two apostrophes (''), and two subtract signs and a greater than sign (→→).

Advanced Uses

If the last text display command of a program is an Output(command, then "Done" will not be displayed

```
PROGRAM:EXAMPLE
:►1rHome
:Output(3,3,"EXAMPLE V1.1
:Output(5,2,"BY TIBASICDEV
```

Command Summary

Displays an expression on the home screen starting at a specified row and column. Wraps around if necessary.

Command Syntax

`Output(row, column, expression)`

Menu Location

While editing a program press:

1. PRGM to enter the PRGM menu
2. RIGHT to enter the I/O menu
3. 6 to choose Output(, or use arrows

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

as the program finishes. Some programmers use this to get rid of the Done message by using an empty Output(command at the end (there is no text after the quote):

```
:Output(1,1,"
```

Since the text displayed by an Output(command wraps, a single command can be used to overwrite the entire screen, by displaying $8 \times 16 = 128$ characters of text starting from row 1, column 1. Since every space on the screen is overwritten, this does not require a ClrHome to clear previously displayed characters. Keep in mind that exactly 16 characters will be on each line.

Optimization

Output(does not allow for more than one expression to be displayed by a single command. However, if several strings are going to be displayed next to each other by several commands, they might be combined into one (keep in mind how wrapping works):

```
:Output(3,3,"Some Text Here  
:Output(4,3,"More Text Here  
can be  
:Output(3,3,"Some Text Here      More Text Here
```

In addition, if you are displaying text on the entire home screen, you can place the all the text in a string and then simply display the string. This is especially useful when combined with movement because you can shift the screen quite easily.

```
:Output(1,1,Str1
```

Command Timings

The Output(command is the fastest possible way of displaying text (short of storing text to a picture and then recalling it). In particular, when going for speed, it should be preferred instead of Disp.

Error Conditions

- **ERR:DOMAIN** is thrown when the starting row or column are not integers in the valid range (this is affected by split screen mode).
- **ERR:INVALID** occurs if this statement is used outside a program.
- An error is **not** thrown when the argument is an empty list (unlike with Disp or pretty much anything else, really)

Related Commands

- Disp
- Text(
- Pause

The Param Command

The Param command enables parametric graphing mode.

Parametric mode is in many ways a generalization of function mode. Instead of writing y as a function of x , both x and y are written as a function of a parameter t (hence the name, parametric mode). You can easily see that equations in function mode are just a special case of equations in parametric mode: if you set x equal to t , then writing $y=f(t)$ is equivalent to writing $y=f(x)$. Of course, graphing a function this way on a calculator will be slightly slower than doing it in function mode directly, because of the overhead.

Parametric mode allows you the greatest freedom of all the possible graphing modes - nearly every curve you could encounter can be expressed in parametric form.

In mathematics, the parameter t is commonly allowed to take on all values from negative to positive infinity. However, this would be impossible to do on a calculator, since the equation would never stop graphing (unlike function mode, there's no easy way to check for which values of t the equation will go off the screen and there's no need to graph it). Instead, the calculator has window variables T_{\min} , T_{\max} , and T_{step} : it will evaluate the parameter at every value from T_{\min} to T_{\max} , increasing by T_{step} each time, and 'connect the dots'.

Polar mode, which you'll read about in the next section, is also a special case of parametric mode:

To graph $r=f(\theta)$, you can instead graph $x=f(t)\cos(t)$ and $y=f(t)\sin(t)$, with t graphed over the same interval as θ .

Advanced Uses

The window variables that apply to parametric mode are:

- **T_{\min}** — Determines the minimum T -value graphed for equations.
- **T_{\max}** — Determines the maximum T -value graphed for equations.
- **T_{step}** — Determines the difference between consecutive T -values.
- **X_{\min}** — Determines the minimum X -value shown on the screen.
- **X_{\max}** — Determines the maximum X -value shown on the screen.
- **X_{scl}** — Determines the horizontal space between marks on the X -axis in AxesOn mode or dots in GridOn mode.
- **Y_{\min}** — Determines the minimum Y -value shown on the screen.
- **Y_{\max}** — Determines the maximum Y -value shown on the screen.
- **Y_{scl}** — Determines the vertical space between marks on the Y -axis in AxesOn mode or

```
Plot1 Plot2 Plot3
X1T=sin(T)
Y1T=cos(T)
X2T=
Y2T=
X3T=
Y3T=
X4T=
```

Command Summary

Enables parametric graphing mode.

Command Syntax

Param

Menu Location

While editing a program, press:

1. MODE to access the mode menu.
2. Use arrows to select Par.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

dots in GridOn mode.

Related Commands

- [Func](#)
- [Polar](#)
- [Seq](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/param>

The Pause Command

The Pause command is used for suspending the execution of a program at a certain point. This is useful when you have text or instructions on the home screen that you want the user to read before the program continues on to the next thing. While the program is paused, the pause indicator turns on in the top-right corner of the screen (it is the dotted line that moves around).

After the user is done reading the text or instructions, they must press ENTER to resume program execution. One place the Pause command is commonly used is right before clearing the screen with [ClrHome](#), because otherwise the text on the screen will show up for a split second before it is erased. The Pause command gives the user ample time to look at and read the text.

```
: Pause
```

An alternative to the Pause command that is commonly used is a [Repeat](#) loop with a [getKey](#) command as the condition. This is sometimes more appropriate in a program if you don't want to bring the program to a complete standstill, and you want the user to be able to resume program execution with any key instead of just ENTER (see [usability](#) for more information).

```
: Repeat getKey  
: End
```

```
PROGRAM:EXAMPLE  
:Disp "I AM KING  
:Pause "PRESS ENTER■
```

Command Summary

Pauses the program until the user presses ENTER.

Command Syntax

Pause or text//

Menu Location

While editing a program, press:

1. PRGM to enter the PRGM menu
2. 8 to choose Pause, or use arrows

Calculator Compatibility

TI-83/84/+SE

Token Size

1 byte

Advanced Uses

The Pause command has an optional argument that can either be text, a number, a variable, or an expression. This argument will be displayed on the next available blank line on the home screen while the program is paused, and it can be scrolled if it is larger than the screen.

Although the Pause command can be used with the graph screen, the argument will still be displayed on the home screen.

Caution: Unlike any other text command, or indeed any other command at all, this optional argument will be stored to Ans after the pause! This could be used to your advantage, but most of the time, it's a nuisance, and if you use Ans for optimization, watch out for this side effect.

Displaying text with the Pause command follows the same pattern as the Disp command, so text is displayed on the left and everything else is displayed on the right. It also means that if there is already text on the seventh row, it will automatically move everything up one row so it can display its text. In addition, the Pause command is affected by the Output(command and its text.

```
PROGRAM:PAUSE
:ClrHome
:"World!
:Disp " Hello "+Ans
:Output(2,2,"Goodbye
:Pause Ans
```

When the calculator is paused, it is possible for another linked calculator to use the GetCalc(command to transfer a variable.

Optimization

When you have a Disp command before a Pause command, you can take the text or variable from the Disp command and place it after the Pause command as its optional argument. This allows you to remove the Disp command. If the Disp command has multiple arguments, you just take the last one off and put it as the optional argument.

```
:Disp A
:Pause
can be
:Pause A
```

When using the optional argument of Pause, it is stored to Ans, and this can in rare cases be used for optimization. The most common one would probably be using Pause to show work for a calculation, as in the following program:

```
:Disp "A+B=
:Pause A+B
:Disp "(A+B)^2=
:Pause Ans^2
:Disp "(A+B)^2-C^2=
:Pause Ans-C^2
```

Error Conditions

- ERR:INVALID occurs if this statement is used outside a program.

Related Commands

- Disp
- Output(
- Text(

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/pause>

The Pen Tool

The Pen tool allows you to draw on the graph screen using a + cursor for the "pen", similar to what you see when accessing the graph screen with graphs or with the Input command. You can find Pen by pressing [2nd][PRGM] to go to the DRAW menu, but it is only accessible from the calculator's home screen (where you do math and run programs); it won't show up when you are inside the program editor.

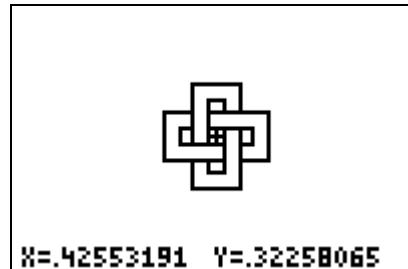
You use the arrow keys to move the pen around, and you can press [2nd][<] and [2nd][>] to move five pixels over instead of just the typical one (this does not work with the up or down keys, since that would change the calculator's contrast). Pressing ENTER starts or stops the drawing respectively, and you just have to press CLEAR twice to return back to the home screen.

When you are done drawing, the calculator stores the coordinates of the cursor to the respective graph screen variables (R and θ for PolarGC format, otherwise X and Y).

Unfortunately, anything that you draw with Pen will be erased or overwritten whenever another program accesses the graph screen or somebody graphs something, so you should store it to a picture if you want to keep it for future use.

Even more unfortunately, you can't erase pixels with Pen. If you're the type of person who makes mistakes once in a while, it might be better to go to the graph screen and choose Pt-Change(from the DRAW menu. This will require you to press ENTER for every pixel you want to draw, but it will also allow you to erase a pixel (by drawing to it again).

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/pen>



Command Summary

Allows the user to draw on the graph screen.

Command Syntax

Pen isn't a command, it's a menu option.

Menu Location

From the home screen, press:

1. 2nd PRGM to enter the DRAW menu
2. Scroll up to select Pen, or use arrows.

Calculator Compatibility

TI-83/84/+SE

The Plot#(Commands

The commands Plot1(), Plot2(), and Plot3(), which are identical except for which stat plot (1, 2, or 3) they affect, define their corresponding stat plot. When the stat plot is defined, it is also turned on so no PlotsOn command is necessary.

The first argument of the commands is always the type of plot, and is one of Scatter, xyLine, Histogram, Boxplot, ModBoxplot, and NormProbPlot - these types are found in the TYPE submenu of the stat plot menu. The other arguments vary. For all but Histogram and Boxplot, there is a *mark* argument - this is a dot, a cross, or a box, symbols that can be found in the MARK submenu of the stat plot menu.

Scatter plot

Plot#(Scatter, *x-list*, *y-list*, *mark*) defines a scatter plot. The points defined by *x-list* and *y-list* are plotted using *mark* on the graph screen.

x-list and *y-list* must be the same length.

xyLine plot

Plot#(xyLine, *x-list*, *y-list*, *mark*) defines an xyLine plot. Similarly to a scatter plot, the points defined by *x-list* and *y-list* are plotted using *mark* on the graph screen, but with an xyLine plot they are also connected by a line, in the order that they occur in the lists.

x-list and *y-list* must be the same length.

Histogram plot

Plot#(Histogram, *x-list*, *freq list*) defines a Histogram plot. The x-axis is divided into intervals that are Xscl wide. A bar is drawn in each interval whose height corresponds to the number of points in the interval. Points that are not between Xmin and Xmax are not tallied.

Xscl must not be too small - it can divide the screen into no more than 47 different bars.

Box plot

Plot#(Boxplot, *x-list*, *freq list*) defines a box plot. A rectangular box is drawn whose left edge is Q_1 (the first quartile) of the data, and whose right edge is Q_3 (the third quartile). A vertical segment is drawn within the box at the median, and 'whiskers' are drawn from the box to the minimum and maximum data points.

PROGRAM: EXAMPLE
:Plot1(BoxPlot,L
1:
:DispGraph■

Command Summary

Displays a statistical plot of one of six types.

Command Syntax

Plot#(*type*,...)

The syntax varies based on the type of plot:

Plot#(Scatter, *x-list*, *y-list*, *mark*)

Plot#(xyLine, *x-list*, *y-list*, *mark*)

Plot#(Histogram, *x-list*, *freq list*)

Plot#(Boxplot, *x-list*, *freq list*)

Plot#(ModBoxplot, *x-list*, *freq list*, *mark*)

Plot#(NormProbPlot, *data list*, *data axis*, *mark*)

Menu Location

While editing a program, press:

1. 2nd PLOT to access the stat plot menu.
2. 1, 2, or 3 (or use arrows) to select Plot1(), Plot2(), Plot3() respectively.

(outside a program, this brings you to the plot editor screen)

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

The box plot ignores the Ymax and Ymin dimensions of the screen, and any plots that aren't box plots or modified box plots. Each box plot takes approximately 1/3 of the screen in height, and if more than one are plotted, they will take up different areas of the screen.

Modified box plot

`Plot#(ModBoxplot, x-list, freq list, mark)` defines a modified box plot. This is almost entirely like the normal box plot, except that it also draws outliers. Whiskers are only drawn to the furthers point within 1.5 times the interquartile range ($Q_3 - Q_1$) of the box. Beyond this point, data points are drawn individually, using *mark*.

The box plot ignores the Ymax and Ymin dimensions of the screen, and any plots that aren't box plots or modified box plots. Each box plot takes approximately 1/3 of the screen in height, and if more than one are plotted, they will take up different areas of the screen.

Normal probability plot

`Plot#(NormProbPlot, data list, data axis, mark)` defines a normal probability plot. The mean and standard deviation of the data are calculated. Then for each point, the number of standard deviations it is from the mean is calculated, and the point is plotted against this number using *mark*. *data axis* can be either X or Y: it determines whether the value of a point determines it's x-coordinate or y-coordinate.

The point behind this rather convoluted process is to test the extent to which the data is normally distributed. If it follows the normal distribution closely, then the result will be close to a straight line - otherwise it will be curved.

Advanced Uses

After doing a regression, a scatter plot of _RESID against the x-list is a useful measure of the effectiveness of the regression. If the plot appears random, this is a good sign; if there is a pattern to the plot, this means it's likely that a better regression model exists.

Optimization

The _ symbol at the beginning of list names can be omitted everywhere in this command.

In addition, every element except the plot type and the data list or data lists are optional, and take on the following default values:

- *freq list* is 1 by default, meaning that all frequencies are 1.
- *mark* is the box by default.
- *data axis* is X by default.

Error Conditions

- **ERR:DIM MISMATCH** is thrown if the x and y lists, or the data and frequency lists, have different dimensions.
- **ERR:STAT** is thrown if *Xscl* is too small in the case of a Histogram.

All errors are thrown when plotting the stat plot, as opposed to when the command is executed, and do not provide a 2:Goto option.

Related Commands

- [Select\(](#)
- [PlotsOn](#)
- [PlotsOff](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/plotn>

The PlotsOff Command

By itself, the command will turn off all three stat plots.

If it is given arguments, there can be any number of them (actually, no more than 255, but this won't stop most people), but they must all be numbers 1 to 3. Then, the command will only turn off the specified plots. Unlike some commands, it is okay to give PlotsOff an expression as an argument (for example, PlotsOff X), as long as it has a value of 1, 2, or 3.

Error Conditions

- **ERR:DOMAIN** is thrown if a plot that is not 1, 2, or 3 is specified.

Related Commands

- [Plot1\(](#), [Plot2\(](#), [Plot3\(](#)
- [PlotsOn](#)
- [Select\(](#)



Command Summary

Turns stat plots (all of them, or only those specified) off.

Command Syntax

PlotsOff numbers//

Menu Location

Press:

1. 2nd PLOT to access the stat plot menu.
2. 4 to select PlotsOff, or use arrows and ENTER.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/plotoff>

The PlotsOn Command

By itself, the command will turn on all three stat plots.

If it is given arguments, there can be any number of them (actually, no more than 255, but this won't stop most people), but they must all be numbers 1 to 3. Then, the command will only turn on the specified plots. Unlike some commands, it is okay to give `PlotsOn` an expression as an argument (for example, `PlotsOn X`), as long as it has a value of 1, 2, or 3.

Error Conditions

- **ERR:DOMAIN** is thrown if a plot that is not 1, 2, or 3 is specified.

Related Commands

- [Plot1\(](#), [Plot2\(](#), [Plot3\(](#)
- [PlotsOff](#)
- [Select\(](#)

```
STAT PLOTS
1:Plot1...Off
  L1 1
2:Plot2...Off
  L1 L2
3:Plot3...Off
  L1 L2
4:PlotsOff
```

Command Summary

Turns stat plots (all of them, or only those specified) on.

Command Syntax

`PlotsOn numbers//`

Menu Location

Press:

1. 2nd PLOT to access the stat plot menu.
2. 5 to select `PlotsOn`, or use arrows and ENTER.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/plotson>

The Pmt_Bgn Command

The `Pmt_Bgn` and `Pmt_End` commands toggle a setting with the finance solver. In `Pmt_Bgn` mode, the calculator assumes that the payments are made at the beginning of each time period, rather than at the end.

Make sure to set the calculator to one of the modes before using the finance solving commands in a program, since otherwise the result is unpredictable.

Related Commands

- [Pmt_End](#)

```
N=0
I%=0
PV=0
PMT=0
FV=0
P/Y=1
C/Y=1
PMT:END [BEGIN]
```

Command Summary

Sets the TVM solver to use payments at the beginning of a period.

Command Syntax

Pmt_Bgn

Menu Location

On the TI-83, press:

1. 2nd FINANCE to access the finance menu.
2. ALPHA F to select Pmt_Bgn, or use arrows and ENTER.

On the TI-83+ or higher, press:

1. APPS to access the applications menu.
2. ENTER to select Finance...
3. ALPHA F to select Pmt_Bgn, or use arrows and ENTER.

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/pmt-bgn>

The Pmt_End Command

The Pmt_End and Pmt_Bgn commands toggle a setting with the finance solver. In Pmt_End mode, the calculator assumes that the payments are made at the end of each time period, rather than at the beginning.

Make sure to set the calculator to one of the modes before using the finance solving commands in a program, since otherwise the result is unpredictable.

Related Commands

- Pmt_Bgn

```
N=0  
I%  
PV=0  
PMT=0  
FV=0  
P/Y=1  
C/Y=1  
PMT:END BEGIN
```

Command Summary

Sets the TVM solver to use payments at the end of a period.

Command Syntax

Pmt_End

Menu Location

On the TI-83, press:

1. 2nd FINANCE to access the finance menu.
2. ALPHA E to select Pmt_End, or use arrows and ENTER.

On the TI-83+ or higher, press:

1. APPS to access the applications menu.
2. ENTER to select Finance...
3. ALPHA E to select Pmt_End, or use arrows and ENTER.

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/pmt-end>

The poissoncdf(Command

This command is used to calculate Poisson distribution cumulative probability. In plainer language, it solves a specific type of often-encountered probability problem, that occurs under the following conditions:

1. A specific event happens at a known average rate (X occurrences per time interval)
2. Each occurrence is independent of the time since the last occurrence
3. We're interested in the probability that the event occurs at most a specific number of times in a given time interval.

The poissoncdf(command takes two arguments: The *mean* is the average number of times the event will happen during the time interval we're interested in. The *value* is the number of times we're interested in the event happening (so the output is the probability that the event happens at most *value* times in the interval). Note that you may need to convert the mean so that the time intervals in both cases match up. This is done by a simple proportion: if the event happens 10 times per minute, it happens 20 times per two minutes.

For example, consider point on a city street where

```
Poissoncdf(1,0  
.3678794412  
Poissoncdf(1,1  
.7357588824  
Poissoncdf(1,3  
.9810118431
```

Command Summary

Calculates the Poisson cumulative probability for a single value

Command Syntax

`poissoncdf(mean, value)`

Menu Location

Press:

1. 2ND DISTR to access the distribution menu
2. ALPHA C to select poissoncdf(, or use arrows.

an average of 5 cars pass by each minute. What is the probability that in a given minute, no more than 3 cars will drive by?

1. The event is a car passing by, which happens at an average rate of 5 occurrences per time interval (a minute)
2. Each occurrence is independent of the time since the last occurrence (we'll assume this is true, though traffic might imply a correlation here)
3. We're interested in the probability that the event occurs at most 3 times in the time interval.

The syntax in this case is:

```
:poissoncdf(5,3)
```

This will give about .265 when you run it, so there's a .265 probability that in a given minute, no more than 3 cars will drive by.

Formulas

The poissoncdf(command can be seen as a sum of poissonpdf(commands:

$$\text{poissoncdf}(\lambda, k) = \sum_{i=0}^k \text{poissonpdf}(\lambda, i) = \sum_{i=0}^k \frac{e^{-\lambda} \lambda^i}{i!} \quad (1)$$

We can also write the poissoncdf(command in terms of the incomplete gamma function:

$$\text{poissoncdf}(\lambda, k) = \frac{\Gamma(k+1, \lambda)}{k!} \quad (2)$$

Related Commands

- [binompdf\(\)](#)
- [binomcdf\(\)](#)
- [poissonpdf\(\)](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/poissoncdf>

Press ALPHA D instead of ALPHA C on a TI-84+/SE with OS 2.30 or higher.

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

This command is used to calculate Poisson

The poissonpdf(Command

distribution probability. In plainer language, it solves a specific type of often-encountered probability problem, that occurs under the following conditions:

1. A specific event happens at a known average rate (X occurrences per time interval)
2. Each occurrence is independent of the time since the last occurrence
3. We're interested in the probability that the event occurs a specific number of times in a given time.

The `poissonpdf(` command takes two arguments: The *mean* is the average number of times the event will happen during the time interval we're interested in. The *value* is the number of times we're interested in the event happening (so the output is the probability that the event happens *value* times in the interval).

For example, consider point on a city street where an average of 5 cars pass by each minute. What is the probability that in a given minute, 8 cars will drive by?

1. The event is a car passing by, which happens at an average rate of 5 occurrences per time interval (a minute)
2. Each occurrence is independent of the time since the last occurrence (we'll assume this is true, though traffic might imply a correlation here)
3. We're interested in the probability that the event occurs 8 times in the time interval

The syntax in this case is:

```
:poissonpdf(5,8
```

This will give about .065 when you run it, so there's a .065 probability that in a given minute, 8 cars will drive by.

Formulas

The value of `poissonpdf(` is given by the formula

$$\text{poissonpdf}(\lambda, k) = \frac{e^{-\lambda} \lambda^k}{k!} \quad (1)$$

Related Commands

- [binompdf\(](#)
- [binomcdf\(](#)

```
PoissonPdf(1,0  
.3678794412  
PoissonPdf(1,1  
.3678794412  
PoissonPdf(1,2  
.1839397206
```

Command Summary

Calculates the Poisson probability for a single value

Command Syntax

`poissonpdf(mean, value)`

Menu Location

Press:

1. 2ND DISTR to access the distribution menu
2. ALPHA B to select `poissonpdf(`, or use arrows.

Press ALPHA C instead of ALPHA B on a TI-84+/SE with OS 2.30 or higher.

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

- [poissoncdf\(](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/poissonpdf>

The ► Polar Command

The ►Polar command can be used when displaying a complex number on the home screen, or with the Disp and Pause commands. It will then format the number as though re^θi mode were enabled. It also works with lists.

```
i
i
i►Polar
1e^(1.570796327i)
{1,i}►Polar
{1 1e^(1.570796327i)}
```

It will also work when displaying a number by putting it on the last line of a program by itself. It does **not** work with Output(, Text(, or any other more complicated display commands.

To actually separate a number into the components of polar form, use abs(and angle(.

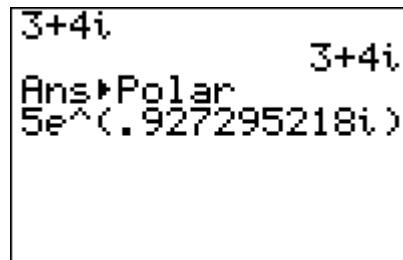
Error Conditions

- **ERR:SYNTAX** is thrown if the command is used somewhere other than the allowed display commands.
- **ERR:DATA TYPE** is thrown if the value is real.

Related Commands

- [►Frac](#)
- [►Dec](#)
- [►Rect](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/polar-display>



Command Summary

Formats a complex value in polar form when displaying it.

Command Syntax

value►Polar

Menu Location

Press:

1. MATH to access the math menu.
2. RIGHT RIGHT to access the CPX submenu.
3. 7 to select ►Polar, or use arrows and ENTER.

Calculator Compatibility

TI-83/84/+/SE

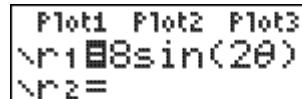
Token Size

2 bytes

The Polar Command

The Polar command enables the polar graphing mode.

Unlike the previous modes, polar mode doesn't use



the more common (x,y) coordinates. Instead, the coordinates (r,θ) are used, where θ is the counterclockwise angle made with the positive x-axis, and r is the distance away from the origin (the point (0,0)). Although it's possible to translate from one system to the other, polar coordinates are more useful for some expressions (and, of course, less useful for others).

In particular, they're very good at graphing anything circle-related. The equation for a circle in polar mode is just $r=1$ (or any other number, for a circle of different radius).

Like in parametric mode, the parameter θ uses the window variables θ_{min} , θ_{max} , and θ_{step} to determine which points are graphed. A common situation is $\theta_{\text{min}}=0$, $\theta_{\text{max}}=2\pi$: in Radian mode, this corresponds to going all the way around the circle. Of course, you could use Degree mode and set θ_{max} to be 360, but this is uncommon in mathematics.

Advanced Uses

The window variables that apply to polar mode are:

- **θ_{min}** — Determines the minimum θ-value graphed for equations.
- **θ_{max}** — Determines the maximum θ-value graphed for equations.
- **θ_{step}** — Determines the difference between consecutive θ-values.
- **X_{min}** — Determines the minimum X-value shown on the screen.
- **X_{max}** — Determines the maximum X-value shown on the screen.
- **X_{scl}** — Determines the horizontal space between marks on the X-axis in AxesOn mode or dots in GridOn mode.
- **Y_{min}** — Determines the minimum Y-value shown on the screen.
- **Y_{max}** — Determines the maximum Y-value shown on the screen.
- **Y_{scl}** — Determines the vertical space between marks on the Y-axis in AxesOn mode or dots in GridOn mode.

Related Commands

- Func
- Param
- Seq

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/polar-mode>

```
\r3=\n\r4=\n\r5=\n\r6=
```

Command Summary

Enables polar graphing mode.

Command Syntax

Polar

Menu Location

While editing a program, press:

1. MODE to access the mode menu.
2. Use arrows to select Pol.

Calculator Compatibility

TI-83/84/+SE

Token Size

1 byte

The PolarGC ("Polar Grid Coordinates") command (like its opposite, the RectGC) command, affects how the coordinates of a point on the graph screen



are displayed. When PolarGC is enabled, the coordinates of a point are displayed as (R,θ).

The polar coordinates of a point can be interpreted as the distance R it is away from the origin (0,0), and the direction θ. θ is the angle that a ray to the point would make with the positive X-axis (so polar coordinates are affected by Degree/Radian mode). An angle of 0 means the point is to the left of the origin; an angle of 90° ($\pi/2$ radians) means it's up from the origin, and so on. So, for example, the point with R=2 and θ=270° ($3\pi/2$ radians) would be two units down from the origin.

Of course, coordinates are only displayed with the CoordOn setting; however, with CoordOff, RectGC and PolarGC are still useful, because in a variety of cases, the coordinates of a point are also stored to variables. PolarGC doesn't change the fact that they're stored to X and Y, as with RectGC; however, with PolarGC, they are also stored to R and θ.

Although the PolarGC command naturally goes with Polar graphing mode, the two settings are independent; you can use both PolarGC and RectGC with any graphing mode.

Advanced

The following situations involve storing coordinates of a point to variables:

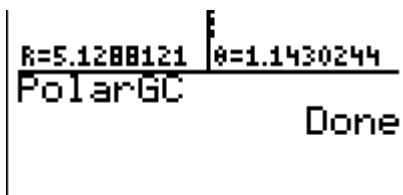
- Graphing an equation
- Tracing an equation or plot
- Moving the cursor on the graph screen
- Using the interactive mode of one of the 2nd DRAW commands
- Using one of DrawF, DrawInv, or Tangent(
- Anything in the 2nd CALC menu.

Naturally, any command like Input or Select(which involves the above, will also store coordinates of a point.

Related Commands

- PolarGC
- CoordOn
- CoordOff

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/polargc>



Command Summary

Sets the calculator to display point coordinates using polar coordinates.

Command Syntax

PolarGC

Menu Location

Press:

1. 2nd FORMAT to access the graph format screen
2. Use arrows and ENTER to select PolarGC.

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

The prgm Command

The prgm command is used to execute a program

from inside another program (at any time while the program is running), with the secondary program acting as a subprogram for that program. Although they are listed in the program menu and can be executed independently like any other program, subprograms are primarily designed to do a particular task for the other program.

You insert the prgm command into the program where you want the subprogram to run, and then type (with the alpha-lock on) the program name. You can also go to the program menu to choose a program, pressing ENTER to paste the program name into your program.

```
PROGRAM :MYPROG
:ClrHome
:Output(3,3,"Hello
:prgmWHATEVER
```

When the subprogram name is encountered during a program, the program will be put on hold and program execution will transfer to the subprogram. Once the subprogram is finished, program execution will go back to the program, continuing right after the subprogram name.

Although subprograms can call themselves or other subprograms, this should be done sparingly because it can cause memory leaks if done too much or if the subprogram doesn't return to the parent program.

Branching is local to each program, so you can't use Goto in one program to jump to a Lbl in another program. In addition, all variables are global, so changing a variable in one program affects the variable everywhere else.

Advanced Uses

Each time you call a TI-Basic program, 16 bytes are used to save your place in the original program so you can return to it correctly. This is a small enough amount that you don't have to worry about it, unless you're low on RAM or use a lot of recursion.

Error Conditions

- ERR:ARCHIVED if the program is archived.
- ERR:SYNTAX, with no 2:Goto option, if the program is an assembly program.
- ERR:UNDEFINED if the program doesn't exist.

See Also

- Subprograms

PRGM EXAMPLE

Done

Command Summary

Calls another program from within a program, with program execution moving to that program.

Command Syntax

prgm*NAME*

Menu Location

Outside the editor, press:

1. PRGM to enter the PRGM menu
2. Use arrows to choose program

When editing a program, press:

1. PRGM to enter the PRGM menu
2. LEFT to enter the EXEC submenu
3. select a program

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

- [Branching](#)
- [Variables](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/prgm>

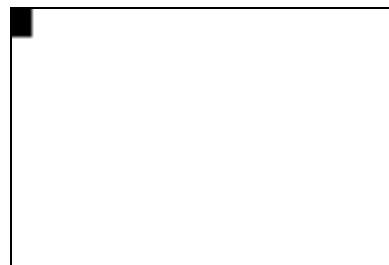
The PrintScreen Command

This command is probably the vestigial remnant of a planned function that wasn't implemented. A token is set aside for it, but the command doesn't actually do anything, and will cause an error if you try to use it. It's not accessible through any menus, though, so that's okay.

The only potential use is to save on memory if you ever need to display "PrintScreen" somewhere - you can display this token instead.

Error Conditions

- **ERR:INVALID** is thrown when trying to use this command.



Command Summary

This command doesn't exist. A token for it does, though.

Command Syntax

PrintScreen

Menu Location

This command requires a hex editor to access.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/printscreen>

The prod(Command

The prod(command calculates the product of all or part of a list.

When you use it with only one argument, the list, it multiplies all the elements of the list. You can also give it a bound of *start* and *end* and it will only multiply the elements starting and ending at those indices (inclusive).

```
prod((1,2,3,4,5
      120
prod((1,2,3,4,5)
      ,3,5
      60
```

```
prod({1,2,3,4,5})  
120  
prod({1,2,3,4,5},2,4)  
24  
prod({1,2,3,4,5},3)  
60
```

Optimization

If the value of `end` is the last element of the list, it can be omitted:

```
prod({1,2,3,4,5},3,5)  
can be  
prod({1,2,3,4,5},3)
```

Error Conditions

- **ERR:DOMAIN** if the starting or ending value aren't positive integers.
- **ERR:INVALID DIM** if the starting or ending value exceed the size of the list, or are in the wrong order.

Related Commands

- [sum\(](#)
- [dim\(](#)
- [seq\(](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/prod>

The Prompt Command

The Prompt command is the simplest way of getting user input on the home screen (getting user input on the graph screen is only possible with the getKey command). Prompt displays variables one per line, with an equal sign and question mark (=?) displayed to the right of each variable. After the user enters a value or expression for the variables and presses ENTER, the values will be stored to the variables and program execution will resume.

Prompt can be used with every variable, but some of the variables have to be entered in a certain way. If the variable is a string or equation, the user must put quotes ("") around the value; the user must also

Command Summary

Calculates the product of all or part of a list.

Command Syntax

```
prod([list],[start],[end])
```

Menu Location

Press:

1. 2nd LIST to access the list menu.
2. LEFT to access the MATH submenu.
3. 6 to select prod(, or use arrows and ENTER.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

PROGRAM: EXAMPLE
:Prompt A
:Disp "A=",A

Command Summary

Prompts the user to enter values for variables and then stores those values to the variables

put curly braces ({}) around lists and square brackets ([]) around matrices. Of course, ending quotes, braces, and brackets can be left off as usual.

When you use Prompt to input a named list, the `L` in front of the name is dropped (so Prompt `L NAME` will display `NAME=?`). This can be confusing with single-letter names: Prompt `L X` and Prompt `X` both display `X=?`. Further enhancing the confusion, if the user enters a list for Prompt `X`, the list will be stored to `L X` instead.

During the Prompt, the user can press [2nd][MODE] to quit the program immediately.

Advanced Uses

Because simply displaying what variable the value will be stored to does not really tell the user what the variable will be used for, you can put a Disp command before Prompt to give the user some more insight into what an appropriate value for the variable would be. The Prompt command will be displayed one line lower, though, because the Disp command automatically creates a new line after itself. (Of course, you could also just use the Input command.)

```
:Disp "Enter the Score  
:Prompt A
```

Optimizations

When you have a list of Prompt commands (and each one has its own variable), you can just use the first Prompt command and combine the rest of the other Prompt commands with it. You remove the Prompt commands and combine the arguments, separating each argument with a comma. The arguments can be composed of whatever combination of variables is desired.

The advantages of combining Prompt commands are that it makes scrolling through code faster, and it is more compact (i.e. smaller) and easier to write than using the individual Prompt commands. The primary disadvantage is that it is easier to accidentally erase a Prompt command with multiple arguments.

```
:Prompt A  
:Prompt Str1  
Combine the Prompts  
:Prompt A,Str1
```

Error Conditions

- ERR:INVALID occurs if this statement is used outside a program.

Related Commands

values to the variables.

Command Syntax

Prompt `variableA[,variableB,...]`

Menu Location

While editing a program press:

1. PRGM to enter the PRGM menu
2. RIGHT to enter the I/O menu
3. 2 to choose Prompt, or use arrows

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

- [Input](#)
- [getKey](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/prompt>

The P►Rx(Command

P►Rx((polar►rectangular x-coordinate) calculates the x-coordinate of a polar point. Polar coordinates are of the form (r, θ) , where θ is the counterclockwise angle made with the positive x-axis, and r is the distance away from the origin (the point (0,0)). The conversion identity $x=r\cos(\theta)$ is used to calculate P►Rx(.

The value returned depends on whether the calculator is in radian or degree mode. A full rotation around a circle is 2π radians, which is equal to 360° . The conversion from radians to degrees is $\text{angle} * 180/\pi$ and from degrees to radians is $\text{angle} * \pi/180$. The P►Rx(command also accepts a list of points.

```
P►Rx(5, π/4)
      3.535533906
5*cos(π/4)
      3.535533906
P►Rx({1, 2}, {π/4, π/3})
      { .7071067812  1}
```

Advanced Uses

You can bypass the mode setting by using the $^\circ$ (degree) and L (radian) symbols. This next command will return the same values no matter if your calculator is in degrees or radians:

```
P►Rx(1, {π/4r, 60°})
      { .7071067812  .5}
```

Optimization

In most cases P►Rx(r,θ) can be replaced by $r\cos(\theta)$ to save a byte:

```
: P►Rx(5, π/12)
can be
: 5cos(π/12)
```

```
P►Rx(5, 60°
      2.5
P►Rx(5, 0, 10°, 20
      °, 30°
      5 4.924038765 ...
```

Command Summary

P►Rx(calculates the x-value (in Cartesian coordinates) given Polar coordinates.

Command Syntax

$\text{P►Rx}(r, \theta)$

Menu Location

Press:

1. 2nd ANGLE to access the angle menu.
2. 7 to select P►Rx(, or use arrows and ENTER.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

Conversely, complicated expressions multiplied by a cosine factor can be simplified by using $\blacktriangleright Rx(r,\theta)$ instead.

```
: (A+BX) cos (π/5)
can be
:P▶Rx(A+BX, π/5)
```

Error Conditions

- **ERR:DIM MISMATCH** is thrown if two list arguments have different dimensions.
- **ERR:DATA TYPE** is thrown if you input a complex argument.

Related Commands

- P▶Ry(
- R▶Pr(
- R▶Pθ(
- cos(

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/p-rx>

The P▶Ry(Command

$P▶Ry($ (polar to rectangular y-coordinate) calculates the y-coordinate of a polar point. Polar coordinates are of the form (r,θ) , where θ is the counterclockwise angle made with the positive x-axis, and r is the distance away from the origin (the point $(0,0)$). The conversion identity $y=r\sin(\theta)$ is used to calculate $P▶Ry($.

The value returned depends on whether the calculator is in radian or degree mode. A full rotation around a circle is 2π radians, which is equal to 360° . The conversion from radians to degrees is $\text{angle} \cdot 180/\pi$ and from degrees to radians is $\text{angle} \cdot \pi/180$. The $P▶Ry($ command also accepts a list of points.

```
P▶Ry(5, π/4)
3.535533906
5 * sin(π/4)
3.535533906
P▶Ry({1, 2}, {π/4, π/3})
{.7071067812 1.732050808}
```

```
P▶Ry(5, 30°
2.5
P▶Ry({1, 2, 3, 4, 5},
, 30°
{.5 1 1.5 2 2.5})
```

Command Summary

$P▶Ry($ calculates the y-value (in Cartesian coordinates) given Polar coordinates.

Command Syntax

$P▶Ry(r,\theta)$

Menu Location

Press:

1. 2nd ANGLE to access the angle menu.
2. 8 to select $P▶Ry($, or use arrows and ENTER.

Advanced Uses

You can bypass the mode setting by using the $^{\circ}$ (degree) and r° (radian) symbols. This next command will return the same values no matter if your calculator is in degrees or radians:

```
P▶Ry(1,{π/4^r,60°})  
{.7071067812 .8660254038}
```

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

Optimization

In most cases $\text{P▶Ry}(r,\theta)$ can be replaced by $r*\sin(\theta)$ to save a byte:

```
:P▶Ry(5,π/12)  
can be  
:5sin(π/12)
```

Conversely, complicated expressions multiplied by a sine factor can be simplified by using $\text{P▶Ry}(r,\theta)$ instead.

```
:(A+BX)sin(π/5)  
can be  
:P▶Ry(A+BX,π/5)
```

Error Conditions

- **ERR:DIM MISMATCH** is thrown if two list arguments have different dimensions.
- **ERR:DATA TYPE** is thrown if you input a complex argument.

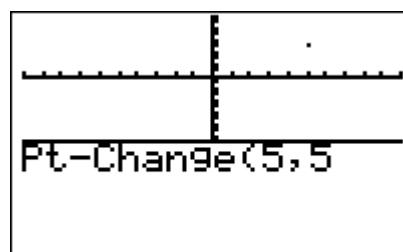
Related Commands

- [P▶Rx\(](#)
- [R▶Pr\(](#)
- [R▶Pθ\(](#)
- [sin\(](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/p-ry>

The Pt-Change(Command

The Pt-Change(command is used to toggle a point (a pixel on the screen) on the graph screen at the given (X,Y) coordinates. If the point is on, it will be turned off and vice versa. Pt-Change(is affected by the window settings, which means you have to change the window settings accordingly, otherwise the point won't show up correctly on the screen.



Pt-Change(can be an interactive command: when on the graph screen, you can select it from the draw menu, and rather than have to input coordinates, be able to draw directly on the screen. Since you can both draw and erase points easily with Pt-Change(, this use of it is often more convenient than the Pen tool.

Related Commands

- [Pt-On\(](#)
- [Pt-Off\(](#)
- [PxI-Change\(](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/pt-change>

The Pt-Off(Command

The Pt-Off(command is used to turn off a point (a pixel on the screen) on the graph screen at the given (X,Y) coordinates. Pt-Off(is affected by the window settings, which means you have to change the window settings accordingly, otherwise the point won't show up correctly on the screen.

Advanced Uses

The Pt-Off(command has an optional third argument that determines the shape of the point (its mark). The mark can be either 1 (dot), 2 (3x3 box), or 3 (3x3 cross). You don't need to specify the mark when using the first mark because it is the default; also, any value that isn't 2 or 3 will be treated as the default of 1.

```
:Pt-Off(5,5,1  
Remove Mark  
:Pt-Off(5,5
```

Command Summary

Toggles a point on the graph screen.

Command Syntax

Pt-Change(X,Y)

Menu Location

While editing a program press:

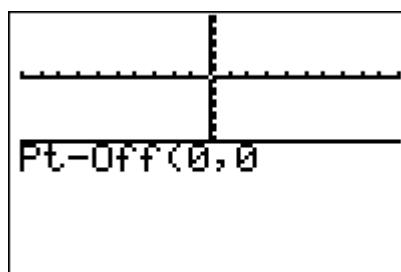
1. 2nd PRGM to enter the DRAW menu
2. RIGHT to enter the POINTS menu
3. 3 to choose Pt-Change

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte



Command Summary

Turns off a point on the graph screen.

Command Syntax

Pt-Off(X,Y[,mark])

Menu Location

While editing a program press:

Related Commands

- [Pt-On\(](#)
- [Pt-Change\(](#)
- [Pxl-Off\(](#)

1. 2nd PRGM to enter the DRAW menu
2. RIGHT to enter the POINTS menu
3. 2 to choose Pt-Off(

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/pt-off>

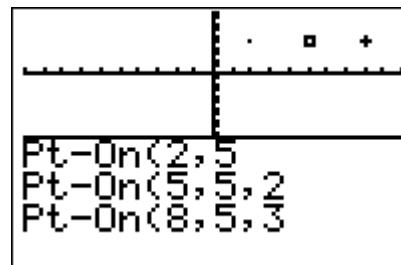
The Pt-On(Command

The Pt-On(command is used to turn on a point (a pixel on the screen) on the graph screen at the given (X,Y) coordinates. Pt-On(is affected by the window settings, which means you have to change the window settings accordingly, otherwise the point won't show up correctly on the screen.

Advanced Uses

The Pt-On(command has an optional third argument that determines the shape of the point (its mark). The mark can be either 1 (dot), 2 (3x3 box), or 3 (3x3 cross). You don't need to specify the mark when using the first mark because it is the default; also, any value that isn't 2 or 3 will be treated as the default of 1. Remember to use the same mark when turning a point off as you used to turn it on.

```
:Pt-On(5,5,1  
Remove Mark  
:Pt-On(5,5
```



Command Summary

Turns on a point on the graph screen.

Command Syntax

Pt-On(X,Y[,*mark*])

Menu Location

While editing a program press:

1. 2nd PRGM to enter the DRAW menu
2. RIGHT to enter the POINTS menu
3. 1 to choose Pt-On(

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

Related Commands

- [Pt-Off\(](#)
- [Pt-Change\(](#)
- [Pxl-On\(](#)

The PwrReg Command

PwrReg tries to fit a power curve ($y=a*x^b$) through a set of points. To use it, you must first store the points to two lists: one of the x-coordinates and one of the y-coordinates, ordered so that the Nth element of one list matches up with the Nth element of the other list. L1 and L2 are the default lists to use, and the List Editor (STAT > Edit...) is a useful window for entering the points.

The calculator does this regression by taking the natural log `ln(` of the x- and of the y-coordinates (this isn't stored anywhere) and then doing a linear regression. The result, $\ln(y)=a*\ln(x)+b$, is transformed into $y=e^{b*x^a}$, which is a power curve. This algorithm shows that if any coordinates are negative or 0, the calculator will instantly quit with `ERR:DOMAIN`.

In its simplest form, PwrReg takes no arguments, and fits a power curve through the points in L1 and L2:

```
: {9,13,21,30,31,31,34→L1  
: {260,320,420,530,560,550,590→L2  
: LnReg
```

On the home screen, or as the last line of a program, this will display the equation of the curve: you'll be shown the format, $y=a*x^b$, and the values of a and b. It will also be stored in the `RegEQ` variable, but you won't be able to use this variable in a program - accessing it just pastes the equation wherever your cursor was. Finally, the statistical variables a, b, r, and r^2 will be set as well. These latter two variables will be displayed only if "Diagnostic Mode" is turned on (see [DiagnosticOn](#) and [DiagnosticOff](#)).

You don't have to do the regression on L1 and L2, but if you don't you'll have to enter the names of the lists after the command. For example:

```
: {9,13,21,30,31,31,34→FAT  
: {260,320,420,530,560,550,590→CALS  
: PwrReg ↵FAT,↵CALS
```

```
PwrReg  
y=a*x^b  
a=2.704220638  
b=4.768743854  
r²=.9991573648  
r=.9995785936
```

Command Summary

Calculates the best fit power curve through a set of points.

Command Syntax

`PwrReg [x-list, y-list, [frequency], [equation]]`

Menu Location

Press:

1. STAT to access the statistics menu
2. LEFT to access the CALC submenu
3. ALPHA A to select PwrReg, or use arrows

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

You can attach frequencies to points, for when a point occurs more than once, by supplying an additional argument - the frequency list. This list does not have to contain integer frequencies. If you add a frequency list, you must supply the names of the x-list and y-list as well, even when they're L1 and L2.

Finally, you can enter an equation variable (such as Y_1) after the command, so that the curve's equation is stored to this variable automatically. This doesn't require you to supply the names of the lists, but if you do, the equation variable must come last. You can use polar, parametric, or sequential variables as well, but since the equation will be in terms of X anyway, this doesn't make much sense.

An example of PwrReg with all the optional arguments:

```
:{9,13,21,30,31,31,34→FAT  
:{260,320,420,530,560,550,590→CALS  
:{2,1,1,1,2,1,1→FREQ  
:PwrReg LFAT,LCA LS,LFREQ,Y1
```

Related Commands

- [LnReg](#)
- [ExpReg](#)
- [SinReg](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/pwrreg>

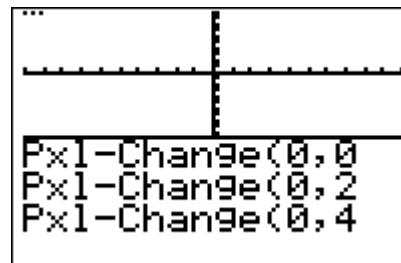
The PxI-Change(Command

The PxI-Change(command is used to toggle the pixel at the given (Y,X) coordinates. If the pixel is on, it will be turned off and vice versa. Please note that the coordinates are switched around so that the row comes first and then the column — it's (Y,X) instead of (X,Y) like the [Pt-Change\(](#) command.

In addition to being easier to use because it is not affected by the window settings (meaning you don't have to set them when using the command), PxI-Change(is faster than its equivalent Pt-Change(command, so it should generally be used instead whenever possible.

Error Conditions

- **ERR:DOMAIN** is triggered if the coordinates are not whole numbers or not in the right range ([0..62] for row, [0..94] for column). These bounds are also affected by split screen mode.



Command Summary

Toggles a pixel on the graph screen.

Command Syntax

`PxI-Change(row,column)`

Menu Location

While editing a program press:

1. 2nd PRGM to enter the

Related Commands

- Pxl-On(
- Pxl-Off(
- pxl-Test(
- Pt-Change(

- DRAW menu
2. RIGHT to enter the POINTS menu
3. 6 to choose Pxl-Change(, or use arrows

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/pxl-change>

The Pxl-Off(Command

The Pxl-Off(command is used to turn off the pixel at the given (Y,X) coordinates. Please note that the coordinates are switched around so that the row comes first and then the column — it's (Y,X) instead of (X,Y) like the Pt-Off(command.

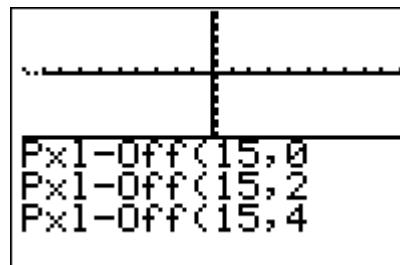
In addition to being easier to use because it is not affected by the window settings (meaning you don't have to set them when using the command), Pxl-Off(is faster than its equivalent Pt-Off(command, so it should generally be used instead whenever possible.

Error Conditions

- ERR:DOMAIN is triggered if the coordinates are not whole numbers or not in the right range ([0..62] for row, [0..94] for column). These bounds are also affected by split screen mode.

Related Commands

- Pxl-On(
- Pxl-Change(
- pxl-Test(
- Pt-Off(



Command Summary

Turns off a pixel on the graph screen.

Command Syntax

Pxl-Off(*row*,*column*)

Menu Location

While editing a program press:

1. 2nd PRGM to enter the DRAW menu
2. RIGHT to enter the POINTS menu
3. 5 to choose Pxl-Off(, or use arrows

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

The Pxl-On(Command

The Pxl-On(command is used to turn on the pixel at the given (Y,X) coordinates. Please note that the coordinates are switched around so that the row comes first and then the column — it's (Y,X) instead of (X,Y) like the Pt-On(command.

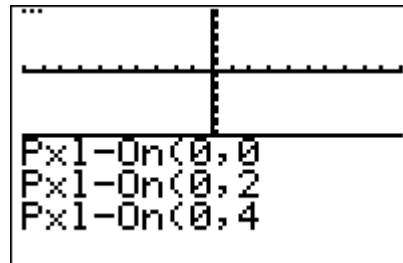
In addition to being easier to use because it is not affected by the window settings (meaning you don't have to set them when using the command), Pxl-On(is faster than its equivalent Pt-On(command, so it should generally be used instead whenever possible.

Error Conditions

- **ERR:DOMAIN** is triggered if the coordinates are not whole numbers or not in the right range ([0..62] for row, [0..94] for column). These bounds are also affected by split screen mode.

Related Commands

- Pxl-Off(
- Pxl-Change(
- pxl-Test(
- Pt-On(



Command Summary

Turns on a pixel on the graph screen.

Command Syntax

Pxl-On(*row*,*column*)

Menu Location

While editing a program press:

1. 2nd PRGM to enter the DRAW menu
2. RIGHT to enter the POINTS menu
3. 4 to choose Pxl-On(, or use arrows

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

The pxl-Test(Command

The pxl-Test(command is used to test a pixel at the

given (Y,X) coordinates of the graph screen, to see whether it is on or off. One is returned if the pixel is on and zero is returned if the pixel is off. Please note that the coordinates are switched around so that the row comes first and then the column — it's (Y,X) instead of (X,Y). This command's coordinates are independent of the window settings.

You can store the result of pxl-Test(to a variable for later use, or use the command in a [conditional](#) or [loop](#).

```
:Pxl-On(25,25
:If pxl-Test(25,25
:Disp "Pixel turned on!
```

Error Conditions

- **ERR:DOMAIN** is triggered if the coordinates are not whole numbers or not in the right range ([0..62] for row, [0..94] for column). These bounds are also affected by split screen mode.

Related Commands

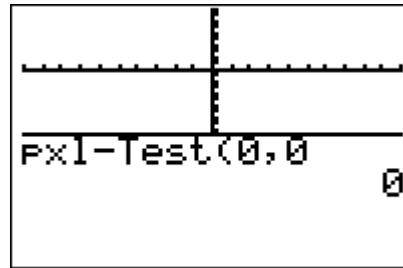
- [Pxl-On\(\)](#)
- [Pxl-Off\(\)](#)
- [Pxl-Change\(\)](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/pxl-test>

The QuadReg Command

The QuadReg command can calculate the best fit quadratic through a set of points. To use it, you must first store the points to two lists: one of the x-coordinates and one of the y-coordinates, ordered so that the Nth element of one list matches up with the Nth element of the other list. L1 and L2 are the default lists to use, and the List Editor (STAT > Edit...) is a useful window for entering the points. You must have at least 3 points, because there's infinitely many quadratics that can go through 2 or 1 point.

In its simplest form, QuadReg takes no arguments, and calculates a quadratic through the points in L1



Command Summary

Tests a pixel on the graph screen to see if it is on or off.

Command Syntax

`pxl-Test(Y,X)`

Menu Location

While editing a program press:

1. 2nd PRGM to enter the DRAW menu
2. RIGHT to enter the POINTS menu
3. 7 to choose pxl-Test(, or use arrows

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

```
QuadReg
y=ax2+bx+c
a=672.9273155
b=-2589.710813
c=2137.51579
R2=.9810530399
```

Command Summary

Calculates the best fit quadratic through a set of points.

and L2:

```
: {9,13,21,30,31,31,34→L1  
: {260,320,420,530,560,550,590→L2  
: QuadReg
```

On the home screen, or as the last line of a program, this will display the equation of the quadratic: you'll be shown the format, $y=ax^2+bx+c$, and the values of a, b, and c. It will also be stored in the RegEQ variable, but you won't be able to use this variable in a program - accessing it just pastes the equation wherever your cursor was. Finally, the statistical variables a, b, c, and R^2 will be set as well. This latter variable will be displayed only if "Diagnostic Mode" is turned on (see [DiagnosticOn](#) and [DiagnosticOff](#)).

You don't have to do the regression on L1 and L2, but if you don't you'll have to enter the names of the lists after the command. For example:

```
: {9,13,21,30,31,31,34→FAT  
: {260,320,420,530,560,550,590→CALS  
: QuadReg ↵FAT,↵CALS
```

You can attach frequencies to points, for when a point occurs more than once, by supplying an additional argument - the frequency list. This list does not have to contain integer frequencies. If you add a frequency list, you must supply the names of the x-list and y-list as well, even when they're L1 and L2.

Finally, you can enter an equation variable (such as Y_1) after the command, so that the quadratic is stored to this equation automatically. This doesn't require you to supply the names of the lists, but if you do, the equation variable must come last. You can use polar, parametric, or sequential variables as well, but since the quadratic will be in terms of X anyway, this doesn't make much sense.

An example of QuadReg with all the optional arguments:

```
: {9,13,21,30,31,31,34→FAT  
: {260,320,420,530,560,550,590→CALS  
: {2,1,1,1,2,1,1→FREQ  
: QuadReg ↵FAT,↵CALS,↵FREQ, Y1
```

Advanced

Note that even if a relationship is actually linear, since a quadratic regression has all the freedom of a linear regression and more, it will produce a better R^2 value, especially if the number of terms is small, and may lead you to (falsely) believe that a relationship is quadratic when it actually isn't. Take the correlation constant with a grain of salt, and consider if the fit is really that much better at the expense of added complexity, and if there's any reason to believe

Command Syntax

QuadReg [*x-list*, *y-list*, [*frequency list*], [*equation variable*]]

Menu Location

Press:

1. STAT to access the statistics menu
2. LEFT to access the CALC submenu
3. 5 to select QuadReg, or use arrows

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

the relationship between the variables may be quadratic.

Related Commands

- [LinReg\(ax+b\)](#)
- [CubicReg](#)
- [QuartReg](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/quadreg>

The QuartReg Command

The QuartReg command can calculate the best fit quartic equation through a set of points. To use it, you must first store the points to two lists: one of the x-coordinates and one of the y-coordinates, ordered so that the Nth element of one list matches up with the Nth element of the other list. L1 and L2 are the default lists to use, and the List Editor (STAT > Edit...) is a useful window for entering the points. You must have at least 5 points, because there's infinitely many quadratics that can go through 4 points or less

In its simplest form, QuartReg takes no arguments, and calculates a quartic through the points in L1 and L2:

```
: {9,13,21,30,31,31,34→L1  
: {260,320,420,530,560,550,590→L2  
: QuartReg
```

On the home screen, or as the last line of a program, this will display the equation of the quartic: you'll be shown the format, $y=ax^4+bx^3+cx^2+dx+e$, and the values of a, b, c, d, and e. It will also be stored in the RegEQ variable, but you won't be able to use this variable in a program - accessing it just pastes the equation wherever your cursor was.

Finally, the statistical variables a, b, c, d, e, and R^2 will be set as well. This latter variable will be displayed only if "Diagnostic Mode" is turned on (see [DiagnosticOn](#) and [DiagnosticOff](#)).

You don't have to do the regression on L1 and L2, but if you don't you'll have to enter the names of the lists after the command. For example:

```
: {9,13,21,30,31,31,34→FAT  
: {260,320,420,530,560,550,590→CALS  
: QuartReg LFACT,LCALS
```

```
QuarticReg  
y=ax4+bx3+cx2+dx+e  
a=29.85572712  
b=-168.2479382  
c=442.7312752  
d=-536.3294433  
e=234.9339766
```

Command Summary

Calculates the best fit quartic equation through a set of points.

Command Syntax

QuartReg [*x-list*, *y-list*, [*frequency list*], [*equation variable*]]

Menu Location

Press:

1. STAT to access the statistics menu
2. LEFT to access the CALC submenu
3. 7 to select QuartReg, or use arrows

Calculator Compatibility

TI-83/84/+SE

Token Size

1 byte

You can attach frequencies to points, for when a point occurs more than once, by supplying an additional argument - the frequency list. This list does not have to contain integer frequencies. If you add a frequency list, you must supply the names of the x-list and y-list as well, even when they're L1 and L2.

Finally, you can enter an equation variable (such as Y_1) after the command, so that the quartic equation is stored to this variable automatically. This doesn't require you to supply the names of the lists, but if you do, the equation variable must come last. You can use polar, parametric, or sequential variables as well, but since the quadratic will be in terms of X anyway, this doesn't make much sense.

An example of QuartReg with all the optional arguments:

```
: {9,13,21,30,31,31,34→FAT  
: {260,320,420,530,560,550,590→CALS  
: {2,1,1,1,2,1,1→FREQ  
: QuartReg ↵ FAT,CALS,FREQ,Y1
```

Advanced

Note that even if a relationship is actually linear, since a quartic regression has all the freedom of a linear regression and much more, it will produce a better R^2 value, especially if the number of points is small, and may lead you to (falsely) believe that a relationship is quartic when it actually isn't. An extreme example is the case of 5 points which are close to being on a line. The linear regression will be very good, but the quartic will seem even better - it will go through all 5 points and have an R^2 value of 1. However, this doesn't make the 5 points special - any 5 (that don't have repeating x-values) will do! Take the correlation constant with a grain of salt, and consider if the fit is really that much better at the expense of much added complexity, and if there's any reason to believe the relationship between the variables may be quartic.

Related Commands

- [LinReg\(ax+b\)](#)
- [QuadReg](#)
- [CubicReg](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/quartreg>

The Radian Command

The Radian command puts the calculator into Radian mode, where the inputs and/or outputs to trig functions are assumed to be radian angles.

Angles measured in radians range from 0 to 2π . They are defined as the arc length of the arc, on a unit circle (circle with radius 1), that corresponds to the angle when it is placed in the center. This definition actually only differs from degree

```
Degree:sin⁻¹(.5  
Radian:sin⁻¹(.5  
30  
.5235987756
```

measurements by a constant factor.

To convert from a degree angle to a radian angle, multiply by $180/\pi$. To go the other way, and get a radian angle from a degree angle, multiply by $\pi/180$.

The following commands are affected by whether the calculator is in Radian or Degree mode:

The input is differently interpreted:

- P►Rx(, P►Ry(
- sin(, cos(, tan(

The output is differently expressed:

- angle(
- R►Pθ(
- sin⁻¹(, cos⁻¹(, tan⁻¹(
- ►Polar (and complex numbers when in re^θi mode)
- r, °

However, some commands are notably unaffected by angle mode, even though they involve angles, and this may cause confusion. This happens with the SinReg command, which assumes that the calculator is in Radian mode even when it's not. As a result, the regression model it generates will graph incorrectly in Degree mode.

Also, complex numbers in polar form are an endless source of confusion. The angle(command, as well as the polar display format, are affected by angle mode. However, complex exponentials (see the e^(command), defined as $e^{i\theta} = \cos\theta + i\sin\theta$, are evaluated as though in Radian mode, regardless of the angle mode. This gives mysterious results like the following:

```
Degree:re^θi
      Done
e^(πi)
      1e^(180i)
Ans=e^(180i)
      0 (false)
```

Overall, it's better to put your calculator in Radian mode when dealing with polar form of complex numbers, especially since no mathematician would ever use degrees for the purpose anyway.

Optimization

It's sometimes beneficial to use the r command instead of switching to Radian mode. The r symbol will make sure a number is interpreted as a radian angle, even in degree mode, so that, for example:

```
Degree
.....Done
sin(π)
.....0548036651
```

Command Summary

Puts the calculator in Radian mode.

Command Syntax

Radian

Menu Location

While editing a program, press:

1. MODE to access the mode menu.
2. Use arrows and ENTER to select Radian.

Calculator Compatibility

TI-83/84/+SE

Token Size

1 byte

$\sin(\pi^r)$
.....
 θ

This is smaller when only one trig calculation needs to be done. Also, it doesn't change the user's settings, which are good to preserve whenever possible.

Related Commands

- [Degree](#)
- [π](#)
- [°](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/radian-mode>

The randBin(Command

`randBin(n,p)` generates a pseudorandom integer between 0 and *n* inclusive according to the binomial distribution $B(n,p)$ - that is, *n* trials of an event with probability of success *p* are performed, and the number of successes is returned.

`randBin(n,p,simulations)` performs the above calculation *simulations* times, and returns a list of the results. The expected (average) result is $n*p$.

n should be an integer greater than or equal to 1, while *p* should be a real number between 0 and 1 inclusive.

`seed→rand` affects the output of `randBin(`

```
0→rand  
0  
randBin(5,1/2  
2  
randBin(5,1/2,10  
{3 3 2 4 3 2 2 2 4 3}
```

```
0→rand  
0  
randBin(100,.5  
50  
randBin(100,.5,4  
{59 50 50 49})
```

Command Summary

Generates a random number with the binomial distribution.

Command Syntax

`randBin(n,p,# simulations)`

Menu Location

Press:

1. MATH to access the [math](#) menu.
2. LEFT to access the PRB submenu.
3. 7 to select `randBin(`, or use arrows.

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

Formulas

The value of `randBin(` for a given seed can be expressed in terms of [rand](#):

$$\text{randBin}(N,P)=\text{sum}(P>\text{rand}(N$$

This is identical to the output of `randBin(` in the sense that for the same seed, both expressions will generate the same random numbers.

Related Commands

- [rand](#)
- [randInt\(](#)
- [randNorm\(](#)
- [randM\(](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/randbin>

The rand Command

rand generates a uniformly-distributed pseudorandom number (this page and others will sometimes drop the pseudo- prefix for simplicity) between 0 and 1. rand(n) generates a list of n uniformly-distributed pseudorandom numbers between 0 and 1. seed→rand seeds (initializes) the built-in pseudorandom number generator. The factory default seed is 0.

L'Ecuyer's algorithm is used by TI calculators to generate pseudorandom numbers.

```
0→rand  
0  
rand  
.9435974025  
rand(2)  
{.908318861 .1466878292}
```

Note: Due to specifics of the random number generating algorithm, the smallest number possible to generate is slightly greater than 0. The largest number possible is actually 1, but since returning a result of 1 would mess up the output of [randBin\(](#) and [randNorm\(](#), the actual value returned in such cases is 1-1.11e-12 (which is displayed as 1, and is "equal" to 1 for the purposes of the \equiv command).

Advanced Uses

To seed the random number generator, store a positive integer to rand (the command will ignore any decimals, and the sign of the number). Seeding the random number generator has several uses:

When writing a program that uses random numbers, you may add a 0→rand instruction to the beginning of the program — this ensures that the program's actions will be repeatable, making it easier to fix a bug. Just don't forget to take it out when you've

```
0→rand  
0  
rand  
.9435974025  
rand(5  
{.908318861 .14...
```

Command Summary

Generates a random number between 0 and 1, or a list of such numbers. Can also be used to set the random number seed.

Command Syntax

rand

rand(# of numbers)

seed→rand

Menu Location

Press:

1. MATH to access the [math](#) menu.
2. LEFT to access the PRB submenu.
3. ENTER to select rand.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

finished writing the program.

Seeding the random number generator can also be used to create fairly secure (unbreakable without a computer) encryption. Pick a secret key, and store it to rand as a seed. Then, perform some randomly generated manipulations on the data you want to encode — for example, shifting each character of a string by a random number. Decoding the message is simple: store the secret key to rand and perform the opposite of those random operations. However, this is impossible to do if you don't know the secret key.

Since generating random numbers is a fairly time-consuming operation, the `rand(# of numbers)` syntax is very effective at generating a delay in your program — just add the line:

```
:rand(N)
```

The bigger N is, the longer the delay. In relation to the commonly used For(loop delay, the number used in the `rand(` delay is about 10 times smaller. However, this code has a side effect of storing a list of random numbers to Ans, which may be undesirable. To avoid this, use this somewhat longer line:

```
:If dim(rand(N))
```

Despite the presence of an If statement, you don't have to worry about the next line being skipped, since `dim(rand(N))` will always be true.

Error Conditions

- **ERR:DOMAIN** if you try to generate a list of random numbers and the list length isn't an integer 1-999.

Related Commands

- [randInt\(](#)
- [randBin\(](#)
- [randNorm\(](#)
- [randM\(](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/rand>

The `randInt(` Command

`randInt(min,max)` generates a uniformly-distributed pseudorandom integer between *min* and *max* inclusive. `randInt(min,max,n)` generates a list of *n* uniformly-distributed pseudorandom integers between *min* and *max*.

seed→rand affects the output of `randInt(`.

```
0→rand
0
randInt(1,10
10
randInt(1,10,4
{10 2 6 5}
```

0→rand

Command Summary

```

0
randInt(1,10)
10
randInt(1,10,5)
{10 2 6 5 8}

```

Optimization

When the lower bound of `randInt()` is 0, you can replace it with `int(#rand)` to save space. For example:

```

:randInt(0,12
can be
:int(13rand

```

Similarly, if you don't want to include zero in the range, you can use a variant of `1-#int(#rand)`:

```
:1-2int(2rand
```

In this particular example, the only values that you will ever get are -1 or 1.

Formulas

The value of `randInt()` for a given seed can be expressed in terms of `rand`:

`randInt(A,B)=`

- when $A < B$, $A + \text{int}((B-A+1)\text{rand})$
- otherwise, $B + \text{int}((A-B+1)\text{rand})$

This is identical to the output of `randInt()` in the sense that for the same seed, both expressions will generate the same random numbers.

Related Commands

- [rand](#)
- [randBin\(\)](#)
- [randNorm\(\)](#)
- [randM\(\)](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/randint>

Command Summary

Generates a random integer between *min* and *max*, inclusive, or a list of such numbers.

Command Syntax

`randInt(min,max[,# of numbers])`

Menu Location

Press:

1. MATH to access the [math](#) menu.
2. LEFT to access the PRB submenu.
3. 5 to select `randInt()`, or use arrows.

Calculator Compatibility

TI-83/84/+SE

Token Size

2 bytes

The `randM()` Command

`randM(M,N)` generates an *M* by *N* matrix whose entries are pseudorandom integers between -9 and

`0→rand`

9 inclusive.

seed→rand affects the output of randM(.

```
0→rand
0
randM(3,3)
[[9 -3 -9]
 [4 -2 0]
 [-7 8 8]]
```

If you actually cared about the bounds of the random numbers, this command would not be very useful, since it's hard to manipulate the matrix to yield uniformly spread random numbers in a different range.

Formulas

The entries of randM(are actually the outputs of successive calls to randInt(-9,9), filled in starting at the bottom right and moving left across each row from the last row to the first.

Error Conditions

- **ERR:INVALID DIM** is thrown if the number of rows or columns of the matrix isn't an integer 1-99.

Related Commands

- [rand](#)
- [randInt\(](#)
- [randNorm\(](#)
- [randBin\(](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/randm>

The randNorm(Command

randNorm(μ, σ) generates a normally-distributed pseudorandom number with mean μ and standard deviation σ . The result returned will most probably be within the range $\mu \pm 3\sigma$. randNorm(μ, σ, n) generates a list of n normally-distributed pseudorandom numbers with mean μ and standard deviation σ .

seed→rand affects the output of randNorm(.

```
randM(3,4
[[9 6 -6 9]
 [-3 -9 4 -2]
 [0 -7 8 8]])
```

Command Summary

Creates a matrix of specified size with the entries random integers from -9 to 9.

Command Syntax

randM(# rows, # columns)

Menu Location

Press:

1. MATRIX (TI-83) or 2nd MATRIX (TI-83+ or higher) to access the matrix menu
2. RIGHT to access the MATH submenu.
3. 6 to select randM(, or use arrows.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

```
0→rand
0
randNorm(0,1
-1.585709623
randNorm(0,1,5
{-1.330473604 1...])
```

```

θ→rand
θ
randNorm(θ,1)
-1.585709623
randNorm(θ,1,3)
{-1.330473604 1.05074514 -.0368

```

Although a theoretical normally distributed variable could take on any real value, numbers on a calculator have a limited precision, which leads to a maximum range of approximately $\mu \pm 7.02\sigma$ for values of randNorm().

Optimization

When the mean is 0 and the standard deviation 1, invNorm(rand) and invNorm(rand(N)) save space over randNorm(0,1) and randNorm(0,1,N) respectively.

Formulas

The value of randNorm(for a given seed can be expressed in terms of rand:

$$\text{randNorm}(\mu, \sigma) = \mu - \sigma \text{invNorm}(\text{rand})$$

This is identical to the output of randNorm(in the sense that for the same seed, both expressions will generate the same random numbers.

Related Commands

- rand
- randInt(
- randBin(
- randM(

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/randnorm>

The real(Command

real(z) returns the real part of the complex number z. If z is represented as x+iy where x and y are both real, real(z) returns x. Also works on a list of complex numbers.

```

real(3+4i)
3

```

Command Summary

Generates a random normally-distributed number with specified mean and standard deviation.

Command Syntax

`randNorm(μ,σ,[n])`

Menu Location

Press:

1. MATH to access the math menu.
2. LEFT to access the PRB submenu.
3. 6 to select randNorm(, or use arrows.

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

real(3+4i)	3
real(i)	0

Advanced Uses

The `real(` command is expanded by several assembly libraries (such as xLIB and Omnicalc) to call their own routines. If xLib is installed, then `real(` will no longer work as intended even in programs that want to use it for its intended purpose.

If you actually want to take the real part of a complex number, and want the program to work with one of these assembly libraries, you could use the `imag(` command instead - `real(Z)` is equivalent to `imag(Z)`. Alternatively, you could tell people using your program to uninstall xLIB or Omnicalc first.

If a program you downloaded has an error and 2:Goto takes you to a line with `real(` and a bunch of arguments, this is probably because the program uses Omnicalc or xLIB which you don't have installed.

Related Commands

- `abs(`
- `angle(`
- `conj(`
- `imag(`

See Also

- Assembly Libraries

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/real-func>

The Real Command

The Real command puts the calculator in real number-only mode. This shouldn't be taken quite literally, as you can still type in i to get complex numbers, and do operations with them (they will be displayed as in a+bi mode, in that case). However, any operation done with **real** numbers that comes out to a complex result, such as taking the square root of a negative number, will throw a ERR:NONREAL ANS error.

There is no real advantage to using Real mode over a+bi mode — it just adds another error condition that wouldn't be triggered otherwise. However, it is the default setting, and so there's a good chance that the calculator will be in Real mode when someone

Command Summary

Returns the real part of a complex value.

Command Syntax

`real(value)`

Menu Location

Press:

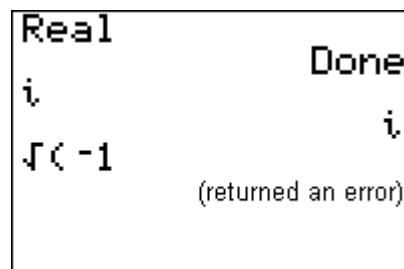
1. MATH to access the math menu.
2. RIGHT, RIGHT to access the CPX (complex) submenu.
3. 2 to select `real(`, or use arrows.

Calculator Compatibility

TI-83/84/+SE

Token Size

2 bytes



Command Summary

Enables real number only mode.

Command Syntax

runs your program. Thus, when using complex numbers implicitly (such as in a quadratic equation solver) you should do something about this.

Advanced Uses

Rather than switch to a+bi mode, you might want to force the calculations to use complex numbers by making the original argument complex. The general way to do this is by adding +0i to the number. However, there may be an optimization in any particular case. See the [quadratic formula](#) routine for a good example of this.

```
Real
      Done
√(-1)
      (causes an error)
√(-1+0i)
      i
```

Real

Menu Location

While editing a program, press:

1. MODE to access the mode menu.
2. Use arrows and ENTER to select Real.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

Related Commands

- [a+bi](#)
- [re^θi](#)

See Also

- [Quadratic Formula](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/real-mode>

The RecallGDB Command

The RecallGDB command recalls graph settings a GDB (Graph DataBase) variable, one of GDB1, GDB2, ..., GDB0 (as indicated by the argument). These settings can be stored to a GDB using the [StoreGDB](#) command.

The settings stored in a GDB include:

- The [graphing mode](#) currently enabled.
- All [equations](#) in the current graphing mode, but NOT other graphing modes.
- All [window variables](#) applicable to the current graphing mode. This does not include zoom variables, table settings, or irrelevant variables such as Tmin when in function mode.
- The menu settings relevant to graphing

```
StoreGDB 1      Done
AxesOff         Done
RecallGDB 1     Done
```

Command Summary

Recalls graph settings from a GDB (Graph DataBase) variable

Command Syntax

(everything in the 2nd FORMAT menu, as well as Connected/Dot and Sequential/Simul settings in the MODE menu)

The number passed to RecallGDB must be one of 0 through 9. It has to be a number: RecallGDB X will not work, even if X contains a value 0 through 9.

Advanced Uses

The StoreGDB and RecallGDB variables are useful in cleaning up after a program finishes running, preserving the user's settings. If your program heavily relies on the graph screen, it may end up editing window size or other graph settings, which the user might want to be saved. This is easily done:

Add StoreGDB 1 (or any other number) to the beginning of your program.

Then, feel free to edit any graph settings you like.

At the end of your program, add RecallGDB 1, followed by DelVar GDB1, to recall the graph settings stored at the beginning.

GDBs can also be useful in adding extra string storage. You can store strings to the Yn variables, and back them up in a GDB; to retrieve them later, recall the GDB and use Equ▶String(to store the equations to the strings again.

Error Conditions

- ERR:DATA TYPE is thrown if the argument is not a *number* 0 through 9.
- ERR:UNDEFINED is thrown if the requested GDB does not exist.

Related Commands

- StoreGDB

See Also

- Cleaning up

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/recallgdb>

The RecallPic Command

RecallPic draws a saved picture to the graph screen (to save a picture, draw it on the graph screen, then save it with StorePic). If something is already drawn on the graph screen, RecallPic will draw new pixels where needed, but it will not erase anything. As a result, you often want to ClrDraw before recalling a

RecallGDB *number*

Menu Location

Press:

1. 2nd DRAW to access the draw menu.
2. LEFT to access the STO menu.
3. 4 to select RecallGDB, or use arrows and ENTER.

Calculator Compatibility

TI-83/84/+SE

Token Size

1 byte



picture.

The number passed to RecallPic must be one of 0 through 9. It has to be a number: RecallPic X will not work, even if X contains a value 0 through 9.

Advanced Uses

A combination of StorePic and RecallPic can be used to maintain a background over which another sprite moves:

1. Draw the background, and save it to a picture file with StorePic.
2. Next, draw the sprite to the screen.
3. When you want to move the sprite, erase it, then use RecallPic to draw the background again.
4. Then draw the sprite to its new location on the screen again (this can be done before or after using RecallPic).

Also, if a screen in your program takes more than a second to draw, and is displayed several times, you might want to consider storing it to a picture the first time it's drawn, and then recalling it every next time you want to draw it.

Error Conditions

- **ERR:DATA TYPE** is thrown if the argument is not a **number** 0 through 9.
- **ERR:UNDEFINED** is thrown if the requested picture does not exist.

Related Commands

- [ClrDraw](#)
- [StorePic](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/recallpic>

The ►Rect Command

The ►Rect command can be used when displaying a complex number on the home screen, or with the [Disp](#) and [Pause](#) commands. It will then format the number as though [a+bi](#) mode were enabled, even when it's not. It also works with lists.

```
i►Polar  
1e^(1.570796327i)
```

TI-BASIC ➤RECT

Command Summary

Recalls a saved picture (one of Pic1, Pic2, ..., Pic0) to the graph screen.

Command Syntax

RecallPic *number*

Menu Location

Press:

1. 2nd DRAW to access the draw menu.
2. LEFT to access the STO submenu.
3. 2 to select RecallPic, or use arrows and ENTER.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

3e^(πi/2)►Rect 3i

Ans►Rect
i

It will also work when displaying a number by putting it on the last line of a program by itself. It does **not** work with Output(, Text(, or any other more complicated display commands.

To actually separate a number into the components of rectangular form, use real(and imag(.

Error Conditions

- **ERR:SYNTAX** is thrown if the command is used somewhere other than the allowed display commands.
- **ERR:DATA TYPE** is thrown if the value is real.

Related Commands

- ►Frac
- ►Dec
- ►Polar

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/rect>

The RectGC Command

The RectGC ("Rectangular Grid Coordinates") command (like its opposite, the PolarGC) command, affects how the coordinates of a point on the graph screen are displayed. When RectGC is enabled, the coordinates of a point are displayed as (X,Y).

The X and Y coordinates of a point are interpreted as the horizontal and vertical distance from the origin (the point (0,0)) Up and right are positive directions, while down and left are negative. For example, the point (1,-2) — that is, the point with x-coordinate 1 and y-coordinate -2 — is one horizontal unit right and two horizontal units down from (0,0).

Of course, coordinates are only displayed with the CoordOn setting; however, with CoordOff, RectGC and PolarGC are still useful, because in a variety of cases, the coordinates of a point are also stored to variables. With RectGC enabled, they are stored to X and Y.

Command Summary

Formats a complex value in rectangular form when displaying it.

Command Syntax

value►Rect

Menu Location

Press:

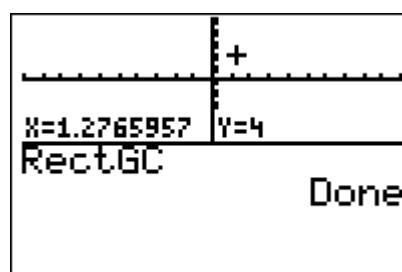
1. MATH to access the math menu.
2. RIGHT RIGHT to access the CPX submenu.
3. 6 to select ►Rect, or use arrows and ENTER.

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes



Command Summary

Sets the calculator to display point coordinates using rectangular (Cartesian) coordinates.

Command Syntax

RectGC

Menu Location

Advanced

The following situations involve storing coordinates of a point to variables:

- Graphing an equation
- Tracing an equation or plot
- Moving the cursor on the graph screen
- Using the interactive mode of one of the 2nd DRAW commands
- Using one of DrawF, DrawInv, or Tangent(
- Anything in the 2nd CALC menu.

Naturally, any command like Input or Select(which involves the above, will also store coordinates of a point.

Related Commands

- PolarGC
- CoordOn
- CoordOff

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/rectgc>

The ref(Command

Given a matrix with at least as many columns as it has rows, the ref(command uses a technique called Gaussian elimination to put the matrix into row-echelon form.

This means that the leftmost N columns (if the matrix has N rows) of the matrix are upper triangular - all entries below the main diagonal are zero. What's more, every entry on the main diagonal is either 0 or 1.

```
[[1,2,5,0][2,2,1,2][3,4,6,2]]  
[[1 2 5 0]  
 [2 2 1 2]  
 [3 4 6 2]]  
ref(Ans)►Frac  
[[1 4/3 2 2/3]  
 [0 1 9/2 -1 ]  
 [0 0 0 0 ]]
```

Press:

1. 2nd FORMAT to access the graph format screen
2. Use arrows and ENTER to select RectGC.

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

```
[[1,2,3][4,5,6]  
 [[1 2 3]  
 [4 5 6]]]  
ref(Ans  
[[1 1.25 1.5]  
 [0 1 2 ]]
```

Command Summary

Puts a matrix into row-echelon form.

Command Syntax

ref(*matrix*)

Menu Location

Press:

1. MATRIX (on the TI-83) or 2nd MATRIX (TI-83+ or higher) to access the matrix menu.
2. RIGHT to access the MATH

Advanced Uses

In theory, a system of linear equations in N variables

can be solved using the `ref` command - an equation of the form $a_1x_1 + \dots + a_nx_n = b$ becomes a row a_1, \dots, a_n, b , and is put into the matrix. If there is a sufficient number of conditions, the last row of the reduced matrix will give you the value of the last variable, and back-substitution will give you the others.

In practice, it's easier to use `rref` instead for the same purpose.

Error Conditions

- **ERR:INVALID DIM** is thrown if the matrix has more rows than columns.

Related Commands

- `rref`
- `rowSwap` and other row operations.

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/ref>

The Repeat Command

A Repeat loop executes a block of commands between the Repeat and End commands until the specified condition is true. The condition is tested at the end of the loop (when the End command is encountered), so the loop will always be executed at least once. This means that you sometimes don't have to declare or initialize the variables in the condition before the loop.

After each time the Repeat loop is executed, the condition is checked to see if it is true. If it is true, then the loop is exited and program execution continues after the End command. If the condition is false, the loop is executed again.

Advanced Uses

When using Repeat loops, you have to provide the code to break out of the loop (it isn't built into the loop). If there is no code that ends the loop, then you will have an infinite loop. An infinite loop just keeps executing, until you have to manually exit the loop (by pressing the ON key). In the case that you actually want an infinite loop, you can just use 0 as the condition. Because 0 is always false (based on Boolean Logic), the loop will never end.

:Repeat 0

submenu.

3. ALPHA A to select `ref`, or use arrows.

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

```
PROGRAM:EXAMPLE
:Disp "INPUT NO.
0 TO 9
:Repeat 0≤X and
X≤9
:Input X
:End
```

Command Summary

Loops through a block of code until the condition is true. Always loops at least once.

Command Syntax

`Repeat condition
statement(s)
End`

Menu Location

While editing a program press:

1. PRGM to enter the PRGM menu
2. 6 to choose Repeat, or use arrows
3. 7 to choose End, or use

```
: statement(s)
: End
```

Each time the program enters a Repeat block, the calculator uses $35 + (\text{size of the condition})$ bytes of memory to keep track of this. This memory is given back to you as soon as the program reaches End. This isn't really a problem unless you're low on RAM, or have a lot of nested Repeat statements. However, if you use Goto to jump out of a Repeat block, you lose those bytes for as long as the program is running — and if you keep doing this, you might easily run out of memory, resulting in ERR:MEMORY.

arrows

Calculator Compatibility

TI-83/84/+SE

Token Size

1 byte

Optimization

The Ans variable (last answer) is a temporary variable that can hold any variable. Ans is changed when there is an expression or variable storage or when pausing with the Pause command. It is mostly useful when you are just manipulating one variable. To use Ans just put an expression on a line by itself; it will automatically be stored to Ans. You can then change the expressions on the next line where the variable was called and put Ans there instead.

Because Repeat loops are executed at least once, you can sometimes put Ans in the condition instead of the variable.

```
:Repeat A
:getKey→A
:End
can be
:Repeat Ans
:getKey→A
:End
```

Command Timings

When deciding whether to use a Repeat loop, as opposed to a For or While loop, it's good to know how Repeat loops stack up against them. This comparison comes from the Code Timings page showing the speeds of the three different kinds of loops:

Format	Bars	Pixels
For(A,0,2000 End	4 bars + 4 pixels	36
Delvar A While A≤2000 A+1→A End	23 bars	184
Delvar A Repeat A>2000 A+1→A End	22 bars + 7 pixels	183

The general conclusion you can take away from this table is that For(loops should be used when speed is a priority, and then you should use Repeat or While loops when the appropriate

circumstance comes up. Each kind of loop has its own place, so it's still good to know how to use all three of them.

Error Conditions

- ERR:INVALID occurs if this statement is used outside a program.

Related Commands

- For(
- While
- If

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/repeat>

The Return Command

When the Return command is used in a program it exits the program (terminating the program execution) and returns the user to the home screen. If it is encountered within loops, the loops will be stopped.

There is some distinction when using Return with subprograms: the Return command will stop the program execution of the subprogram, and program execution will go back to the calling program, continuing right after the subprogram call. If this functionality is not desired, then you should use the Stop command instead. Generally, though, you should use Return instead of Stop.

```
:ClrHome  
:Input "Guess:",A  
:Stop  
Replace Stop with Return  
:ClrHome  
:Input "Guess:",A  
:Return
```

```
PROGRAM:EXAMPLE  
:Disp "THIS GETS  
DONE  
:Return  
:Disp "BUT THIS  
DOESN'T"
```

Command Summary

Stops the program and returns the user to the home screen. If the program is a subprogram, however, it just stops the subprogram and returns program execution to the parent program.

Command Syntax

Return

Menu Location

Press:

1. PRGM to enter the PRGM menu
2. ALPHA SIN to choose Return, or use arrows

Calculator Compatibility

TI-83/84/+/SE

Token Size

Optimization

You don't have to put a Return command at the end of a program or subprogram if you can organize the program so that it just naturally quits. When the calculator reaches the end of a program, it will automatically stop executing as if it had encountered a Return command (the Return is implied).

```
:ClrHome
:Input "Guess:",A
:Return
Remove the Return
:ClrHome
:Input "Guess:",A
```

1 byte

Error Conditions

- **ERR:INVALID** occurs if this statement is used outside a program.

Related Commands

- [prgm](#)
- [Stop](#)

See Also

- [Subprograms](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/return>

The $re^{\theta}i$ Command

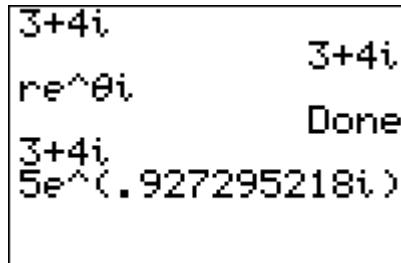
The $re^{\theta}i$ command puts the calculator into polar complex number mode. This means that:

- Taking square roots of negative numbers, and similar operations, no longer returns an error.
- Complex results are displayed in the form $re^{(\theta)i}$ (hence the name of the command)

The mathematical underpinning of this complex number format is due to the fact that if (x,y) is a point in the plane using the normal coordinates, it can also be represented using coordinates (r,θ) where r is the distance from the origin and θ is the angle that the line segment to the point from the origin makes to the positive x-axis (see [Polar](#) and [PolarGC](#) for more information on polar coordinates and graphing). What does this have to do with complex numbers? Simple: if $x+yi$ is a complex number in normal (rectangular) form, and $re^{(\theta)i}$ is the same number in polar form, then (x,y) and (r,θ) represent the same point in the plane.

Of course, that has a lot to do with how you define imaginary exponents, which isn't that obvious.

An equivalent form to polar form is the form $r[\cos(\theta)+i\sin(\theta)]$.



Command Summary

Puts the calculator into $re^{\theta}i$ mode.

Command Syntax

`re^θi`

Menu Location

Press:

1. MODE to access the mode menu.
2. Use the arrow keys and ENTER to select $re^{\theta}i$

Calculator Compatibility

Unfortunately, the calculator seems to have some confusion about the use of degree and radian angle measures for θ in this mode (the answer is: you can only use radians — degrees make no sense with complex exponents). When calculating a value $re^{(\theta i)}$ by using the e^(command and plugging in numbers, the calculator assumes θ is a radian angle, whether it's in Degree or Radian mode. However, when *displaying* a complex number as $re^{(\theta i)}$, the calculator will display θ in radian or degree measure, whichever is enabled. This may lead to such pathological output as:

```
Degree: re^θi
      Done
e^(πi)
      1e^(180i)
Ans=e^(180i)
      θ (false)
```

It's recommended, then, to use Radian mode whenever you're in $re^{(\theta i)}$ mode.

Related Commands

- Real
- a+bi

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/re-thetai>

The round(Command

`round(value,[#decimals])` returns *value* rounded to *#decimals* decimal places. *#decimals* must be < 10 . The default value for *#decimals* is 9. Also works on complex numbers, lists and matrices.

```
round(5.45,0)
      5
round(5.65,0)
      6
round(-5.65,0)
      -6
round(π)-π
      4.102e-10
round(π,4)
      3.1416
round({1.5,2.4,3.8},0)
      {2,2,4}
round([[1.8,3.5,120.3][3,-1,0.2]],0)
      [[2    4      120]
       [3    -1     0      ]]
```

TI-83/84+/SE

Token Size

1 byte

round(π,0)	3
round(π,5)	3.14159
round(π	3.141592654

Command Summary

Truncates a number to a specified number of decimal places.

Command Syntax

`round(value,[#decimals])`

Menu Location

Press:

1. MATH to select the math

Advanced Uses

Sometimes, round-off error will cause the result of an expression to be slightly off of the correct integer value — for example, a result may be 5.00000000013 instead of 5. If the error is small enough, it will not even be visible if you recall the variable on the home screen. However, this is enough to cause a ERR:DOMAIN error with commands such as sub(and Output(, which require their arguments to be integers.

The easiest way to fix this problem is by wrapping the different arguments in a round(instruction. For example, you may replace Output(X,1,">") with Output(round(X),1,">"). The int(command will not work here because the round-off error may be negative, such as 4.9999999986 instead of 5, in which case the number will be rounded down to 4.

Error Conditions

- ERR:DOMAIN if the number of places to round to is not an integer 0 through 9.

Related Commands

- int(
- iPart(
- fPart(

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/round>

The *row(Command

The *row(command multiplies a row of a matrix by a scalar factor and returns the result. It is an elementary row operation used in Gaussian Elimination.

```
[ [1,2][3,4]
  [[1 2]
   [3 4]]
*row(10,Ans,1)
  [[10 20]
   [3 4 ]]
```

```
[[1,2][3,4
      [[1 2]
       [3 4]]]
*row(10,Ans,2
      [[1 2]
       [30 40]])
```

Advanced Uses

You can multiply columns instead of rows with the aid of the T (transpose) command.

menu.

2. RIGHT to select the NUM submenu.
3. 2 to select round(, or use arrows.

Calculator Compatibility

TI-83/84/+SE

Token Size

1 byte

Command Summary

Multiplies a row by a scalar.

Command Syntax

*row(factor,matrix,row)

Menu Location

Error Conditions

- **ERR:INVALID DIM** is thrown if the row argument isn't a valid row (is larger than the size of the matrix, or otherwise bad)

Related Commands

- rowSwap(
- row+(
- *row+(

Press:

1. MATRX (on a TI-83) or 2nd MATRX (TI-83+ or higher) to access the matrix menu.
2. RIGHT to access the MATH submenu.
3. ALPHA E to select *row(+, or use arrows and ENTER.

Calculator Compatibility

TI-83/84/+SE

Token Size

1 byte

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/timesrow>

The *row+(Command

The *row+(adds a multiple of one row to another row and returns the result. It is an elementary row operation used in Gaussian Elimination.

```
[[1,2][3,4]]  
[[1 2]  
 [3 4]]  
*row+(10,Ans,1,2)  
[[3 4 ]  
 [31 42]]
```

```
[[1,2][30,40]]  
[[1 2 ]  
 [30 40]]  
*row+( -1,Ans,1,2  
  
[[1 2 ]  
 [29 38]])
```

Advanced Uses

You can add columns instead of rows with the aid of the T (transpose) command.

Error Conditions

- **ERR:INVALID DIM** is thrown if one of the row arguments isn't a valid row (larger than the matrix size, or otherwise bad)

Related Commands

- rowSwap(
- row+(
- *row(

Command Summary

Adds a multiple of one row of a matrix to another.

Command Syntax

*row+(factor,matrix,row1,row2)

Menu Location

Press:

1. MATRX (on a TI-83) or 2nd MATRX (TI-83+ or higher) to access the matrix menu.
2. RIGHT to access the MATH submenu.
3. ALPHA F to select *row+(, or use arrows and ENTER.

Calculator Compatibility

Token Size

1 byte

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/timesrowplus>

The row+(Command

The `row+(` command adds one row of a matrix to the second, and returns the result. It is an elementary row operation used in Gaussian Elimination.

```
[[1,2][3,4]]
 [[1 2]
 [3 4]]
row+(Ans,1,2)
 [[1 2]
 [4 6]]
```

Advanced Uses

You can add columns instead of rows with the aid of the T ([transpose](#)) command.

Error Conditions

- **ERR:INVALID DIM** is thrown if one of the row arguments isn't a valid row (larger than the matrix size, or otherwise bad)

Related Commands

- [rowSwap\(](#)
- [*row\(](#)
- [*row+\(](#)

```
[[1,2][30,40]]
 [[1 2]
 [30 40]]
row+(Ans,1,2)
 [[1 2]
 [31 42]]
```

Command Summary

Adds one row of a matrix to another.

Command Syntax

`row+(matrix, row1, row2)`

Menu Location

Press:

1. MATRIX (on a TI-83) or 2nd MATRIX (TI-83+ or higher) to access the matrix menu.
2. RIGHT to access the MATH submenu.
3. ALPHA D to select `row+(`, or use arrows and ENTER.

Calculator Compatibility

TI-83/84+/SE

Token Size

1 byte

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/rowplus>

The rowSwap(Command

The `rowSwap(` command swaps two rows of a matrix and returns the result. It is an elementary row operation used in Gaussian Elimination.

```
[[1,2][3,4]
 [[1 2]
 [3 4]]
rowSwap(Ans,1,2)
 [[3 4]
 [1 2]]
```

Advanced Uses

You can swap columns instead of rows with the aid of the T (transpose) command.

Error Conditions

- **ERR:INVALID DIM** is thrown if one of the row arguments isn't a valid row (larger than the matrix size, or otherwise bad)

Related Commands

- `row+()`
- `*row()`
- `*row+()`

```
[[1,2][3,4]
 [[1 2]
 [3 4]]
rowSwap(Ans,1,2)
 [[3 4]
 [1 2]]
```

Command Summary

Swaps two rows of a matrix.

Command Syntax

`rowSwap(matrix, row1, row2)`

Menu Location

Press:

1. MATRIX (on a TI-83) or 2nd MATRIX (TI-83+ or higher) to access the matrix menu.
2. RIGHT to access the MATH submenu.
3. ALPHA C to select `rowSwap()`, or use arrows and ENTER.

Calculator Compatibility

TI-83/84/+SE

Token Size

1 byte

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/rowswap>

The R►Pr(Command

`R►Pr(` (Rectangular to polar radius) takes the (x,y) (Cartesian) coordinates, and gives the radius coordinate r of the same point in (r,θ) (polar) mode. The identity used for this conversion is $r^2 = x^2+y^2$

```
R►Pr(3,4)
 5
```

```
R►Pr(3,4
      5
R►Pr(5,12
      13
```

```

 $\sqrt{(3^2+4^2)}$ 
5
R►Pr({6,5},{8,12})
{10 13}

```

The function works even when the equivalent $\sqrt{x^2+y^2}$ fails due to overflow:

```
R►Pr(3e99,4e99)
5e99
```

Optimization

R►Pr() is the smallest way to implement the distance formula $d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. Just pass it the values $x_1 - x_2$ and $y_1 - y_2$ as arguments:

```
: $\sqrt{((5-2)^2+(4-0)^2)}$ 
can be
:R►Pr(5-2,4-0)
```

Error Conditions

- [ERR:DATA TYPE](#) is thrown if you input a complex argument.
- [ERR:DIM MISMATCH](#) is thrown if two list arguments have different dimensions.

Related Commands

- [P►Rx\(\)](#)
- [P►Ry\(\)](#)
- [R►Pθ\(\)](#)
- [abs\(\)](#)
- [\$\sqrt{}\$](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/r-pr>

The R►Pθ() Command

R►Pθ() (Rectangular to polar θ) takes the (x,y) (Cartesian) coordinate, and returns the angle that the ray from $(0,0)$ to (x,y) makes with the positive x-axis. This is the θ -coordinate of the same point in (r,θ) (polar) mode. The identity used for this conversion is $\tan(\theta)=y/x$, with the correct inverse being chosen depending on the quadrant that the point is in. The range of the angle returned is $-\pi < \theta \leq \pi$. R►Pθ() can also be used on lists.

Command Summary

R►Pr() calculates the radius component (in polar coordinates) given the Cartesian coordinates.

Command Syntax

R►Pr(x,y)

Menu Location

Press:

1. 2nd ANGLE to access the angle menu.
2. 5 to select R►Pr(), or use arrows and ENTER.

Calculator Compatibility

TI-83/84/+SE

Token Size

1 byte

```
R►Pθ(1,1
      .7853981634
R►Pθ({1,2,3,4,
      5
      {.7853981634 1....
```

$\text{R}\blacktriangleright\text{P}\theta()$ is equivalent to the `atan2()` instruction seen in C++ and FORTRAN.

```
R►Pθ(3,4)
.927295218
tan¹(4/3)
.927295218
R►Pθ(0,{1,-1})
{1.570796327, -1.57096327}
```

$\text{R}\blacktriangleright\text{P}\theta()$ is affected by Degree and Radian mode in its output, which is an angle measured in degrees or radians respectively.

Advanced Uses

If you want the result to always be a radian angle, regardless of mode settings, you can divide the result by 1° :

```
R►Pθ(x,y)/1^°
```

If you want the result to always be a degree angle, regardless of mode settings, you can divide the result by 1° :

```
R►Pθ(x,y)/1°
```

Error Conditions

- ERR:DATA TYPE is thrown if you input a complex argument.
- ERR:DIM MISMATCH is thrown if two list arguments have different dimensions.

Related Commands

- P►Rx(
- P►Ry(
- R►Pr(
- angle(
- tan⁻¹(

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/r-ptheta>

Command Summary

$\text{R}\blacktriangleright\text{P}\theta()$ calculates the angle coordinate (in polar coordinates) given the Cartesian coordinates.

Command Syntax

```
R►Pθ(x,y)
```

Menu Location

Press:

1. 2nd ANGLE to access the angle menu.
2. 6 to select $\text{R}\blacktriangleright\text{P}\theta()$, or use arrows and ENTER.

Calculator Compatibility

TI-83/84/+SE

Token Size

1 byte

The r (Radian Symbol) Command

NOTE: Due to the limitations of the wiki markup language, the r command on this page does not appear as it would on the calculator. See [Wiki Markup Limitations](#) for more information.

Normally, when the calculator is in degree mode, the trigonometric functions only return values calculated in degrees. With the r symbol you can have the angle evaluated as if in radian mode because it converts the angle into degrees.

One full rotation around a circle is 2π radians, which is equal to 360° . To convert an angle in radians to degrees you multiply by $180/\pi$, and to convert from degrees to radians multiply by $\pi/180$.

In degree mode:

```
sin(π)          \sine of Pi degrees
.0548036651
sin(π^r)
θ
```

In radian mode:

```
sin(π)
θ
sin(π^r)
θ      \There's no difference
```

Degree	Done
$\sin(\pi/6)r$	$1.594973073e-4$
$2\pi r$	360

Command Summary

If the calculator is in degree mode, the r (radian) symbol converts a radian angle to degrees.

Command Syntax

$angle^r$

Menu Location

Press:

1. 2nd ANGLE to access the angle menu.
2. 3 to select r , or use arrows and ENTER.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

Optimization

When you only call the trig function once in a program and want it calculated in radians, instead of changing the mode you can just use $^\circ$ to save one-byte (the newline from using the command Radian)

```
:Radian
:sin(X)
can be
:sin(X^r)
```

Related Commands

- $^\circ$ (degree symbol)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/radian-symbol>

The rref(Command

Given a matrix with at least as many columns as rows, the rref(command puts a matrix into reduced row-echelon form using Gaussian elimination.

This means that as many columns of the result as possible will contain a pivot entry of 1, with all entries in the same column, or to the left of the pivot, being 0.

```
[[1,2,5,0][2,2,1,2][3,4,6,2]
 [[1 2 5 0]
 [2 2 1 2]
 [3 4 7 3]]
rref(Ans)
 [[1 0 0 6
 [0 1 0 -5.5]
 [0 0 1 1]]]
```

```
[[1,2,3][4,5,6
 [[1 2 3]
 [4 5 6]]
rref(Ans)
 [[1 0 -1]
 [0 1 2]]]
```

Command Summary

Puts a matrix into reduced row-echelon form.

Command Syntax

rref(*matrix*)

Menu Location

Press:

1. MATRIX (on the TI-83) or 2nd MATRIX (TI-83+ or higher) to access the matrix menu.
2. RIGHT to access the math menu.
3. ALPHA B to select rref(, or use arrows and ENTER.

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

Advanced Uses

The rref(command can be used to solve a system of linear equations. First, take each equation, in the standard form of $a_1x_1 + \dots + a_nx_n = b$, and put the coefficients into a row of the matrix.

Then, use rref(on the matrix. There are three possibilities now:

- If the system is solvable, the left part of the result will look like the identity matrix. Then, the final column of the matrix will contain the values of the variables.
- If the system is inconsistent, and has no solution, then it will end with rows that are all 0 except for the last entry.
- If the system has infinitely many solutions, it will end with rows that are all 0, including the last entry.

This process can be done by a program fairly easily. However, unless you're certain that the system will always have a unique solution, you should check that the result is in the correct form, before taking the values in the last column as your solution. The Matr▶list(command can be used to store this column to a list.

Error Conditions

- ERR:INVALID DIM is thrown if the matrix has more rows than columns.

Related Commands

- [ref](#)
- [rowSwap](#)(and other row operations.

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/rref>

The Sci Command

The Sci command puts the calculator in scientific notation mode, so that all results are displayed in scientific notation: as a (possibly fractional) number between 1 and 10 (not including 10) multiplied by a power of 10.

```
Sci
      Done
1000
      1e3
{1, 2, 3}
      {1e0 2e0 3e0}
```

30000	30000
Sci	Done
30000	3e4

Command Summary

Puts the calculator in scientific notation mode.

Command Syntax

Sci

Menu Location

While editing a program, press:

1. MODE to access the mode menu.
2. Use arrows and ENTER to select Sci.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

Related Commands

- [Normal](#)
- [Eng](#)
- [Float](#)
- [Fix](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/sci>

The Select(Command

When Select(is called, if it has any [Scatter](#) or [xyLine](#) plots to work with, it displays the graph screen and allows the user to pick a left bound and then a right bound on one of the plots (the left and right keys

F1:L1,L2	.	.
Left Bound?		x

move from point to point, while the up and down keys switch plots). Then, it stores all the points between those bounds to *x-list name* and *y-list name*. Finally, it sets the chosen plot to use *x-list name* and *y-list name* as its X and Y lists.

Optimization

It isn't necessary to add the `L` symbol before list names:

```
:Select(LX,LY)  
can be  
:Select(X,Y)
```

Error Conditions

- **ERR:INVALID** is thrown if there are no enabled Scatter or xyLine plots for the command to work with.

Related Commands

- [Plot1\(](#), [Plot2\(](#), [Plot3\(](#)
- [Input](#)

X=4 Y=2048.5147
Select(L₃,L₄)

Command Summary

Allows the user to select a subinterval of any enabled Scatter or xyLine plots.

Command Syntax

Select(*x-list name*, *y-list name*)

Menu Location

Press:

1. 2nd LIST to access the list menu.
2. RIGHT to access the OPS submenu.
3. 8 to select Select(, or use arrows and ENTER.

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/select>

The Send(Command

The Send(command is used for sending data to a CBL (Calculator Based Laboratory) device (or another compatible device) via a link cable. With some exceptions, Send('s argument must be a variable: a real number, list, matrix, string, equation, picture, or GDB. An expression or a number will not work — Send(5) or Send([A][B]) is invalid.

The exceptions are list or matrix elements (that is, you can do Send([A](1,1)) or Send(L1(2)) without an error) and non-variable lists typed out with { } brackets and commas.

Related Commands

Send(A Done
Send(L₁ Done

Command Summary

Sends data or a variable to a connected CBL device.

Command Syntax

- [Get\(](#)
- [GetCalc\(](#)

Send(*variable*)

Menu Location

While editing a program, press:

1. PRGM to access the program menu.
2. RIGHT to access the I/O submenu.
3. ALPHA B to select Send(.

Calculator Compatibility

TI-83/84/+SE

Token Size

1 byte

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/send>

The seq(Command

The seq(command is very powerful, as it is (almost) the only command that can create a whole list as output. This means that you will need make use of it almost every time that you use lists. The seq(command creates a list by evaluating a formula with one variable taking on a range of several values.

It is similar in this to the For(command, but unlike For(, instead of running a block of commands, it only evaluates a formula. Like the For(command, there is an optional "step" that you can use to get every 3rd, every 5th, etc. value in the range.

Some sample uses of the command:

```
:seq(I,I,3,7
```

- evaluates the expression 'I' with I taking on the values 3..7
- returns {3,4,5,6,7}

```
:seq(AX2,X,1,7
```

- evaluates the expression 'AX²' with X taking on the values 1..7
- returns {A,4A,9A,16A,25A,36A,49A},

```
seq(I2,I,3,6
      {9 16 25 36}
seq(X,X,7,0, -1
      {7 6 5 4 3 2 1 ...}
```

Command Summary

Creates a list by evaluating a formula with one variable taking on a range of values, optionally skipping by a specified step.

Command Syntax

seq(*formula*, *variable*, *start-value*, *end-value* [, *step*])

Menu Location

While editing a program, press:

1. 2nd LIST to enter the LIST menu
2. RIGHT to enter the OPS submenu
3. 5 to choose seq(or use

depending on the value of A

```
:seq(Y1(T),T,1,9,2
```

- evaluates the expression 'Y1(T)' with T taking on every 2nd value 1..9
- returns {Y1(1),Y1(3),Y1(5),Y1(7),Y1(9)} depending on Y1

5. ⌂ to choose seq(), or use arrows.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

Note: the value of the variable used in the expression does not change. If X has some value stored to it, and you do a seq() command using X, X will still hold that original value. However, if X was undefined before the command, after the command, it will be defined and have a value of 0.

Advanced Uses

The step argument supplied can be negative. If it is, and if the starting value is greater than the ending value, then the sequence will "go backward", evaluating the expression in the opposite order. For example:

```
:seq(I,I,1,7  
    {1,2,3,4,5,6,7}  
:seq(I,I,7,1,-1  
    {7,6,5,4,3,2,1}
```

You can use seq(to get a "sublist", that is, to get a list that is only a section of another list. This is pretty much the only effective way to extract a sublist. For example, to get the 2nd through 10th elements of L1, do the following:

```
:seq(L1(I),I,2,10
```

While using seq(), the calculator can still interpret keypresses and store them to getKey. One possible way you can use this feature is to make a password function that asks the user to enter in the correct password before time expires.

Optimizations

It's faster to do an operation on an entire list, than to do the same operation inside a seq() command. For example, take the following:

```
:seq(Y1(T),T,1,9  
can be  
:Y1(seq(T,T,1,9
```

However, not all commands that work for numbers will work for lists. A notable example is getting an element from a list: L1({1,2,3} will not return the first, second, and third elements of L1, so you will have to put the L1 inside the seq() command.

For this same reason, you shouldn't use a seq() command when you're really performing an operation on each element of a list. For example, if L1 has 10 elements:

```
:seq(L1(I)^2,I,1,10
can be
:L1^2
```

A seq(command can replace a For(command, if all you're doing inside the For(command is storing to an element of a list. This will improve on both speed and size of your program. For example:

```
:For(I,1,10
:I^2→L1(I
:End
can be
:seq(I^2,I,1,10→L1
```

The seq(command itself can often be replaced with an unusual use of the binomcdf(or binompdf(commands, improving speed and sometimes size as well. However, this optimization is fairly advanced; read the pages for those commands to learn about it.

Error Conditions

- **ERR:ILLEGAL NEST** is thrown if you try to use seq(inside of another seq(command.

Related Commands

- For(
- binompdf(
- binomcdf(

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/seq-list>

The Seq Command

The Seq command enables sequence graphing mode.

Sequential mode is used for graphing sequences, which can be thought of as functions from the positive (or non-negative) integers. The TI-83 calculators let n be the independent variable in this situation, and the three sequences, instead of using subscripts, use the letters u, v, and w.

One of the main advantages of sequential mode is that it allows recursive definitions: $u(n)$ can be defined in terms of $u(n-1)$ and $u(n-2)$. For recursive definitions to work, an initial case must be defined: this is done using the variables $u(nMin)$, $v(nMin)$, and $w(nMin)$. The constant $nMin$ is the initial case, for which the calculator will use a specific value

```
Plot1 Plot2 Plot3
nMin=0
··u(n)·u(n-1)+u(n
-2)
u(nMin)·{1,1}
··v(n)=
v(nMin)=
··w(n)=
```

Command Summary

Enables sequence graphing mode.

Command Syntax

Seq

rather than the formula.

For example, say a bunny population starts out at 100 and doubles each year. We can describe this situation using the recursive definition $u(n)=2u(n-1)$ (this just says that the n th year population is twice the population of the previous year); then we set $u(nMin)=100$. Note that without $u(nMin)$, the equation would be meaningless - without the initial population, we have no way to calculate any other population.

When you're using more than one previous value — both $u(n-1)$ and $u(n-2)$ — you need more than one initial value, and then $u(nMin)$ becomes a list.

Advanced Uses

Sequence graphing mode has several submodes that can be selected from the 2nd FORMAT screen. They are Time, Web, uvAxes, uwAxes, and vwAxes. Sequences are still defined in the same way, but these modes control the way that they're graphed.

The window variables that apply to sequence mode are:

- **$nMin$** — Determines the minimum n -value calculated for equations.
- **$nMax$** — Determines the maximum n -value calculated for equations.
- **PlotStart** — Determines the first value of n that is actually graphed.
- **PlotStep** — Determines the difference between consecutive *graphed* values of n .
- **Xmin** — Determines the minimum X-value shown on the screen.
- **Xmax** — Determines the maximum X-value shown on the screen.
- **Xscl** — Determines the horizontal space between marks on the X-axis in AxesOn mode or dots in GridOn mode.
- **Ymin** — Determines the minimum Y-value shown on the screen.
- **Ymax** — Determines the maximum Y-value shown on the screen.
- **Yscl** — Determines the vertical space between marks on the Y-axis in AxesOn mode or dots in GridOn mode.

Related Commands

- Func
- Param
- Polar

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/seq-mode>

The Sequential Command

Puts the calculator into sequential graphing mode

Menu Location

While editing a program, press:

1. MODE to access the mode menu.
2. Use arrows to select Seq.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

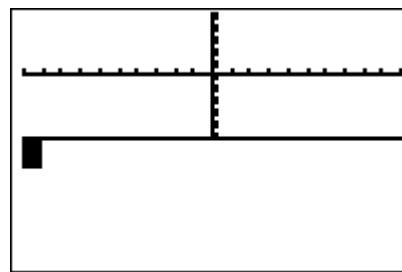
(the default). When multiple equations are enabled at the same time, sequential graphing mode means that they will be graphed one after the other (as opposed to Simul mode, in which they will be graphed simultaneously)

If you use a list in an equation, as with $Y1=\{1,2,3\}X$, this will graph several equations that will always graph separately, regardless of this setting, which only affects multiple functions in different equation variables.

Make sure not to confuse this with Seq mode, which is referred to in this guide as sequence graphing mode.

Related Commands

- Simul



Command Summary

Puts the calculator into sequential graphing mode.

Command Syntax

Sequential

Menu Location

While editing a program, press:

1. MODE to access the mode menu.
2. Use arrows and ENTER to select Sequential.

Calculator Compatibility

TI-83/84/+SE

Token Size

2 bytes

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/sequential>

The setDate(Command

The setDate(command sets the date of the clock on the TI-84+/SE calculators. It takes three arguments: the year, the month, and the day. All three of these must be integers; in particular, year must be four digits, and month and day can be one or two digits. They represent the associated value that goes with a respective date. For example, this would set the date to January 1, 2008:

```
: setDate(2008,1,1
```

Once you have set the date, you can display it in three different formats on the mode screen using the setDtFmt(command: Month/Day/Year,

```
PROGRAM:DATE
:Disp "DATE:", getDate
:setDate(2008,8,8)
:Disp "NEW DATE:", getDate
```

Command Summary

Sets the date of the clock on the TI-84+/SE.

Command Syntax

Day/Month/Year, or Year/Month/Day. Of course, the date will only show up if the clock is on; if you need to turn the clock on, use the ClockOn command or select 'TURN CLOCK ON', displayed in place of the clock on the mode screen.

Related Commands

- getDate
- getDtFmt
- getDtStr(
- setDtFmt(

setDate(year,month,day)

Menu Location

This command can only be found in the catalog. Press:

1. 2nd CATALOG to enter the command catalog
2. s to skip to commands starting with S
3. Scroll down to setDate(and select it

Calculator Compatibility

TI-84+/SE

Token Size

2 bytes

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/setdate>

The setDtFmt(Command

The setDtFmt(command sets the date format of the clock on the TI-84+/SE calculators when displaying the date on the mode screen. There are three different formats available, and you simply use the respective value (can be either a literal number or a variable) to display the desired one: 1 (M/D/Y), 2 (D/M/Y), or 3 (Y/M/D). For example, this would set the date format to Month/Day/Year:

```
:setDtFmt(1
```

In order for the date format to work, you need to set the date using either the setDate(command, or by going into the set clock menu (accessible by pressing ENTER on the 'SET CLOCK' message that is displayed at the bottom of the mode screen). Of course, the date will only show up if the clock is on; if you need to turn the clock on, use the ClockOn command, or scroll down to the 'TURN CLOCK ON' message that is displayed in place of the clock on the mode screen and press ENTER twice.

Related Commands

```
PROGRAM:DATE
:setDtFmt(3)
:Disp "DATE FORM
AT NOW:", "Y/M/D"
```

Command Summary

Sets the date format of the clock on the TI-84+/SE.

Command Syntax

setDtFmt(*value*)

Menu Location

This command can only be found in the catalog. Press:

1. 2nd CATALOG to enter the command catalog
2. s to skip to commands starting with S

- [getDate](#)
- [setDate\(\)](#)
- [getDtFmt](#)
- [getDtStr\(\)](#)

3. Scroll down to setDtFmt(and select it

Calculator Compatibility

TI-84+/SE

Token Size

2 bytes

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/setdtfmt>

The setMode() Command

The setMode() command is used, mainly by programmers, to change mode settings (outside a program, of course, you can just select the settings in the MODE menu). When a setting is changed, it returns the old value of the setting. There are two ways to use the command:

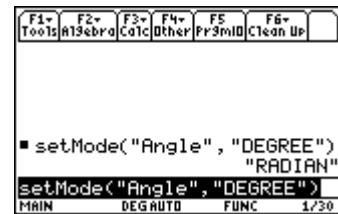
- `setMode(setting,value)` will change *setting* to *value*, and return the old value of *setting*.
- `setMode({set1,val1,...})` will change *set1* to *val1*, *set2* to *val2*, and so on, for any number of settings, and return a list in the same format of settings and their old values.

The first format is used to change only one setting, and the second to change several settings.

Both settings and values are identified by strings (not case-sensitive). All the strings involved are given in the [Table of Mode Settings](#).

An example of setMode():

```
setMode("Angle","DEGREE")
        "RADIAN"
setMode({"Angle","RADIAN",Split Screen
        {"Split Screen" "FULL" "Angle" "
```



Command Summary

Changes one or more mode settings.

Command Syntax

- `setMode(setting,value)`
- `setMode({set1,val1,...})`

Menu Location

From the program editor toolbar:

1. Choose F6 - Mode
2. Press 1-G to select the desired setting's submenu
3. Press 1-? to paste `setMode("setting","value")`

Calculator Compatibility

This command works on all calculators.

Token Size

3 bytes total:

- 0xE3 ('extra command' tag)
- 0x11 (command identifier)

Advanced Uses

Unfortunately, the strings depend on language localization. For [compatibility](#) with other languages, there is an alternate identification for the settings and values: you can use a string containing a number identifying the setting or value (see the

Use the output of the list version of setMode() to restore settings to what they were previously:

```
:setMode({ "Angle", "RADIAN"})→oldmode  
...  
:setMode(oldmode)
```

Error Conditions

- **130 - Argument must be a string** occurs if the data type of arguments is incorrect.
- **260 - Domain error** occurs if the string used to identify a setting is incorrect or misspelled.
- **430 - Invalid for the current mode settings** occurs if a setting depends on other settings that are incorrect (e.g. setting "Split 2 App" if "Split Screen" is "FULL").
- **450 - Invalid in a function or current expression** occurs if setMode() is used in a function.

Related Commands

- [getMode\(\)](#)
- [setGraph\(\)](#)
- [setTable\(\)](#)

See Also

- [Table of Mode Settings](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/68k:setmode>

The setTime(Command

The setTime(command sets the time of the clock on the TI-84+/SE calculators. It takes three arguments: the hour, the minute, and the second. The hour must be in 24 hour format — where 13 is equal to 1 P.M. — and the minute and second need to be a valid number within the appropriate range (1-60). For example, this would set the time to 12:30:30:

```
:setTime(12,30,30)
```

```
PROGRAM:TIME  
:setTime(0,0,0)  
:Disp "TIME:",GetTmStr(GetTmFmt)
```

Command Summary

Sets the time of the clock on the TI-84+/SE.

Once you have set the time, you can display it in two different formats on the [mode screen](#) using the

setTmFmt(command: 12 (12 hour) or 24 (24 hour). Of course, the time will only show up if the clock is on; if you need to turn the clock on, use the ClockOn command, or scroll down to the 'TURN CLOCK ON' message that is displayed in place of the clock on the mode screen and press ENTER twice.

Related Commands

- getTime
- getTmFmt
- getTmStr(
- setTmFmt(

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/settime>

The setTmFmt(Command

The setTmFmt(command sets the time format of the clock on the TI-84+/SE calculators when displaying the time on the mode screen. There are two different formats available, and you simply use the respective value (can be either a literal number or a variable) to display the desired one: 12 (12 hour) or 24 (24 hour). For example, this would set the time format to 24 hour:

```
:setTmFmt(24
```

In order for the time format to work, you need to set the time using either the setTime(command, or by going into the set clock menu (accessible by pressing ENTER on the 'SET CLOCK' message that is displayed at the bottom of the mode screen). Of course, the time will only show up if the clock is on; if you need to turn the clock on, use the ClockOn command, or scroll down to the 'TURN CLOCK ON' message that is displayed in place of the clock on the mode screen and press ENTER twice.

Related Commands

Command Syntax

`setTime(hour,minute,second)`

Menu Location

This command can only be found in the catalog. Press:

1. 2nd CATALOG to enter the command catalog
2. s to skip to commands starting with S
3. Scroll down to setTime(and select it

Calculator Compatibility

TI-84+/SE

Token Size

2 bytes

```
PROGRAM:TIME
:Disp "TIME FORM"
AT:",getTmFmt
:setTmFmt(24)
:Disp "NOW:",get
TmFmt
```

Command Summary

Sets the time format of the clock on the TI-84+/SE.

Command Syntax

`setTmFmt(value)`

Menu Location

This command can only be found in the catalog. Press:

1. 2nd CATALOG to enter the command catalog

RELATED COMMANDS

- [getTime](#)
- [setTime\(](#)
- [getTmFmt](#)
- [getTmStr\(](#)

2. s to skip to commands starting with S
3. Scroll down to setTmFmt(and select it

Calculator Compatibility

TI-84+/SE

Token Size

2 bytes

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/settmfmt>

The SetUpEditor Command

The SetUpEditor command is meant as an auxiliary for the List Editor (which can be accessed by pressing STAT ENTER (Edit...)). The list editor provides a convenient interface for looking at the elements in lists, or editing those elements (especially when the elements of two lists are connected to each other, such as a list for X-coordinates and one for Y-coordinates, since they will be shown in the same row).

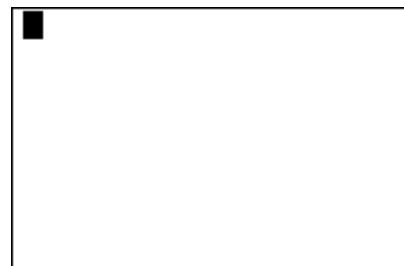
The SetUpEditor command sets which lists are shown in this list editor. By default, it selects the lists L1, L2, L3, L4, L5, and L6. To do this, just use the command with no arguments:

```
: SetUpEditor
```

However, you can use it to select any lists that you have defined, or even lists that you haven't defined yet. To do this, simply put the lists you want as arguments to the command. For example, if you want to edit the lists FOO and BAR, do:

```
: SetUpEditor FOO,BAR
```

The List Editor doesn't do anything when you are running a program, so it may seem as though SetUpEditor is nearly useless in programs. This is not the case, however, because of SetUpEditor's powerful side effect: if the lists it is given as arguments are archived, it will unarchive them. If they don't exist, it will create them with size 0.



Command Summary

Sets which lists are shown in the List Editor. As a side effect, unarchives the lists, and creates them if they don't exist.

By default, works with L1-L6.

Command Syntax

SetUpEditor [*list, list,...*]

Menu Location

Press:

1. STAT to enter the STAT menu.
2. 5 to choose SetUpEditor, or use arrows.

Calculator Compatibility

TI-83/84/+SE

Token Size

Advanced Uses

2 bytes

Due to this side effect, SetUpEditor can be used for lists with external data such as saved games or high scores. When the user first runs the program, the assumption is you don't know anything about the state of those lists: they may be archived, or they may not even exist. You can deal with both of those individually: storing to the dimension will create the list if it didn't exist, and the UnArchive command will move the list to RAM if it wasn't there.

However, if you're wrong about the list, both of these commands will cause an error. If the list exists but is archived, storing to its dimension will cause an ERR:ARCHIVE error. If the list doesn't exist, unarchiving it will cause an ERR:UNDEFINED error. Sounds like a vicious circle.

The SetUpEditor command allows you to deal with both of these problems at once. Say the program saves its data in LSAVE. Use the SetUpEditor command on it, and from then on you know that the list exists AND that it is unarchived.

```
: SetUpEditor SAVE
```

At the end of the program, you should clean up after yourself, though. You don't want the user to see the list SAVE in the editor (he might be tempted to edit it and give himself a huge high score, for one thing). So you should use the SetUpEditor command again, this time without arguments, to reset the editor to its default state.

For more information about using SetUpEditor in the context of saving data, see the page on saving.

Similar Commands

- dim(
- ClrList
- UnArchive

See Also

- Saving Data
- Highscores
- Program Cleanup

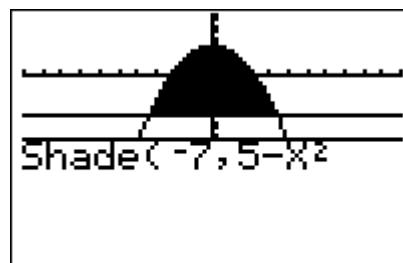
For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/setupeditor>

The Shade(Command

The Shade(command draws two functions and shades the area between them.

Shade(*lower func*, *upper func*, [*xmin*, *xmax*, *pattern #*, *resolution*])

- *lowerfunc* and *upperfunc* are the two functions (whenever *lowerfunc*<*upperfunc*, the area between them will be shaded)



- *xmin* and *xmax* (optional) are left and right boundaries on where to shade.
- *pattern #* (optional) is an integer 1-4 determining which pattern to use:
 - 1 — vertical shading (default)
 - 2 — horizontal shading
 - 3 — diagonal shading (negative slope)
 - 4 — diagonal shading (positive slope)
- *resolution* (optional) is an integer 1-8 determining the spacing between shading lines. When it's 1 (default), everything is shaded, when it's 2, one pixel is skipped between lines, and so on - when it's 8, seven pixels are skipped.

Note that if you don't supply the *resolution* argument, it defaults to 1 and everything gets shaded regardless of the pattern.

Advanced Uses

`Shade(Ymin,Ymax)` is the smallest (though not the fastest) way to shade the entire screen.

Related Commands

- [DrawF](#)
- [DrawInv](#)
- [Tangent\(](#)

See Also

- [Shading Circles](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/shade>

The ShadeF(Command

`ShadeF(` is equivalent to `Fcdf(` in terms of the probability it calculates: if a random variable follows the *F*-distribution, you can use it to calculate the probability that the variable's value falls in a certain interval. However, in addition to calculating the probability, this command also draws the distribution, and shades the interval whose area represents the probability you want.

Note that this command does not actually return the value it calculates in `Ans` or anywhere else: it's merely displayed on the graph. If you're going to use the value in further calculations, you'll have to use `Fcdf(` as well.

Command Summary

Graphs two functions and shades the area between them.

Command Syntax

`Shade(lower func, upper func, [xmin, xmax, pattern #, resolution])`

Menu Location

Press:

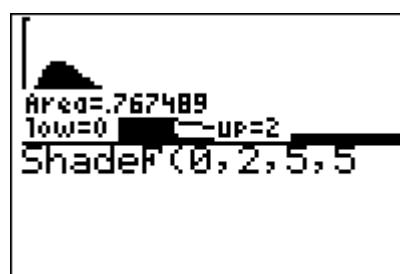
1. 2nd DRAW to access the draw menu.
2. 7 to select Shade(, or use arrows.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte



Command Summary

Finds the probability of an interval of the *F*-distribution, and graphs the distribution with the interval's area

Like Fcdf(), ShadeF() takes four arguments: the lower bound, the upper bound, and the numerator and denominator degrees of freedom.

Advanced

Often, you want to find a "tail probability" - a special case for which the interval has no lower or no upper bound. For example, "what is the probability x is greater than 2?". The TI-83+ has no special symbol for infinity, but you can use E99 to get a very large number that will work equally well in this case (E is the decimal exponent obtained by pressing [2nd] [EE]). Use E99 for positive infinity, and -E99 for negative infinity.

The ShadeF() command's output is affected by the graphing window, and on many windows you won't be able to get a good idea of what the graph looks like. The entire graph is in the first quadrant, so unless you want to see bits of the axes, Ymin and Xmin should be 0. Ymax should probably be around .5 - this would depend on the specific F -distribution of course, as would Xmax.

Keep in mind that ShadeF() is a drawing command and not the graph of an equation, so changing graph settings, the ClrDraw command, and a great deal of other things will erase its output.

Related Commands

- Fpdf()
- Fcdf()
- Shade()

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/shadef>

The ShadeNorm(Command

ShadeNorm() is equivalent to normalcdf() in terms of the probability it calculates: if a random variable follows the normal distribution, you can use it to calculate the probability that the variable's value falls in a certain interval. However, in addition to calculating the probability, this command also draws the normal curve, and shades the interval whose area represents the probability you want.

Note that this command does not actually return the value it calculates in Ans or anywhere else: it's merely displayed on the graph. If you're going to use the value in further calculations, you'll have to use

shaded.

Command Syntax

ShadeF(lower, upper, numerator df, denominator df)

Menu Location

Press:

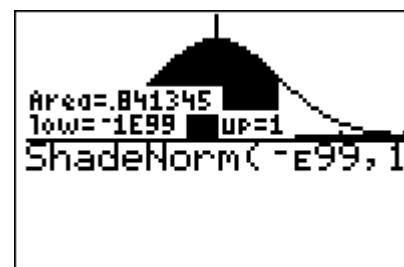
1. 2ND DISTR to access the distribution menu
2. RIGHT to select the DRAW submenu
3. 4 to select ShadeF(), or use arrows

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes



Command Summary

Finds the probability of an interval of the normal curve, and graphs the

normalcrt(as well.

There are two ways to use ShadeNorm(). With two arguments (lower bound and upper bound), the calculator will assume you mean the standard normal distribution, and use that to find the probability corresponding to the interval between "lower bound" and "upper bound". You can also supply two additional arguments to use the normal distribution with a specified mean and standard deviation. For example:

```
for the standard normal distribution  
:ShadeNorm(-1,1
```

```
for the normal distribution with mean  
:ShadeNorm(5,15,10,2.5
```

normal curve with the interval's area shaded.

Command Syntax

ShadeNorm(*lower*, *upper*, μ , σ)

Menu Location

Press:

1. 2ND DISTR to access the distribution menu
2. RIGHT to select the DRAW submenu
3. ENTER to select ShadeNorm(

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

Advanced

Often, you want to find a "tail probability" - a special case for which the interval has no lower or no upper bound. For example, "what is the probability x is greater than 2?". The TI-83+ has no special symbol for infinity, but you can use E99 to get a very large number that will work equally well in this case (E is the decimal exponent obtained by pressing [2nd] [EE]). Use E99 for positive infinity, and -E99 for negative infinity.

It can be hard to find the best window for ShadeNorm() to work in, since it doesn't automatically zoom for you. For the standard curve, the graph doesn't go above $y=.5$ (a good value for Ymax); Ymin should probably be something small. Xmin and Xmax could be -3 to 3 (3 deviations out); change this around to see more or less of the graph.

For nonstandard curves, increasing the standard deviation stretches and flattens the curve; by dividing Ymax and multiplying Xmin and Xmax by the standard deviation, you'll account for this effect. To account for the mean, add it to both Xmin and Xmax. You may also choose to standardize the lower and upper values instead by applying the formula $(z-\mu)/\sigma$.

Keep in mind that ShadeNorm is just a drawing command and not an actual graphed function, so resizing the window, ClrDraw, and a bunch of other things will simply get rid of it.

Related Commands

- normalpdf(
- normalcdf(
- invNorm(
- Shade(

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/shadenorm>

The Shade_t(Command

`Shade_t(` is equivalent to `tcdf(` in terms of the probability it calculates: if a random variable follows the Student's t distribution, you can use it to calculate the probability that the variable's value falls in a certain interval. However, in addition to calculating the probability, this command also draws the distribution, and shades the interval whose area represents the probability you want.

Note that this command does not actually return the value it calculates in `Ans` or anywhere else: it's merely displayed on the graph. If you're going to use the value in further calculations, you'll have to use `tcdf(` as well.

Like `tcdf(`, `Shade_t(` takes three arguments: the lower bound, the upper bound, and the degrees of freedom.

Advanced

Often, you want to find a "tail probability" - a special case for which the interval has no lower or no upper bound. For example, "what is the probability x is greater than 2?". The TI-83+ has no special symbol for infinity, but you can use `E99` to get a very large number that will work equally well in this case (`E` is the decimal exponent obtained by pressing [2nd] [`EE`]). Use `E99` for positive infinity, and `-E99` for negative infinity.

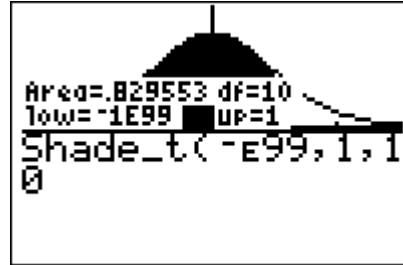
The `Shade_t(` command's output is affected by the graphing window, and on many windows you won't be able to get a good idea of what the graph looks like. For best results, `Ymin` should be either 0 or a small negative number, and `Ymax` should be 0.5 or less. `Xmin` and `Xmax` should be opposites of each other (so the middle of the graph is 0), but how large they are depends on the degrees of freedom and on how much of the graph you want to see: -4 and 4 are good starting places.

Keep in mind that `Shade_t(` is a drawing command and not the graph of an equation, so changing graph settings, the `ClrDraw` command, and a great deal of other things will erase its output.

Related Commands

- [tpdf\(](#)
- [tcdf\(](#)
- [invT\(](#)
- [Shade\(](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/shade-t>



Command Summary

Finds the probability of an interval of the Student's t distribution, and graphs the distribution with the interval's area shaded.

Command Syntax

`Shade_t(lower, upper, df)`

Menu Location

Press:

1. 2ND DISTR to access the distribution menu
2. RIGHT to select the DRAW submenu
3. 2 to select `Shade_t(`, or use arrows

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

The Shadex²(Command

Shadex²(is equivalent to $\chi^2\text{cdf}($ in terms of the probability it calculates: if a random variable follows the χ^2 distribution, you can use it to calculate the probability that the variable's value falls in a certain interval. However, in addition to calculating the probability, this command also draws the χ^2 curve, and shades the interval whose area represents the probability you want.

Note that this command does not actually return the value it calculates in Ans or anywhere else: it's merely displayed on the graph. If you're going to use the value in further calculations, you'll have to use $\chi^2\text{cdf}($ as well.

The Shadex²(command takes three arguments. *lower* and *upper* identify the interval you're interested in. *df* specifies the degrees of freedom (selecting from an infinite family of χ^2 distributions).

Thus, the following code would find the probability of χ^2 between 0 and 1 on a χ^2 distribution with 2 degrees of freedom, and shade this interval:

```
: Shadex2(0,1,2
```

Advanced

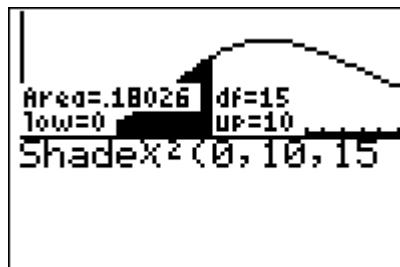
Often, you want to find a "tail probability" - a special case for which the interval has no lower or no upper bound. For example, "what is the probability x is greater than 2?". The TI-83+ has no special symbol for infinity, but you can use E99 to get a very large number that will work equally well in this case (E is the decimal exponent obtained by pressing [2nd] [EE]). Use E99 for positive infinity, and -E99 for negative infinity.

It can be hard to find the best window for Shadex²(to work in, since it doesn't automatically zoom for you. For any number of degrees of freedom (except for 1), the graph doesn't go above $y=.5$ (a good value for Ymax); Ymin should probably be something small and negative. Xmin should be around 0 (possibly slightly less if you like seeing axes), while Xmax probably shouldn't go above 5.

Keep in mind that Shadex²(is just a drawing command and not an actual graphed function, so resizing the window, ClrDraw, and a bunch of other things will simply get rid of it.

Related Commands

- $\chi^2\text{pdf}($



Command Summary

Finds the probability of an interval of the χ^2 distribution, and graphs this distribution with the interval's area shaded.

Command Syntax

Shadex²(*lower*, *upper*, *df*)

Menu Location

Press:

1. 2ND DISTR to access the distribution menu
2. RIGHT to select the DRAW submenu
3. 3 to select Shadex²(, or use arrows

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

- x^2 cdf(
- Shade(

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/shadechisquare>

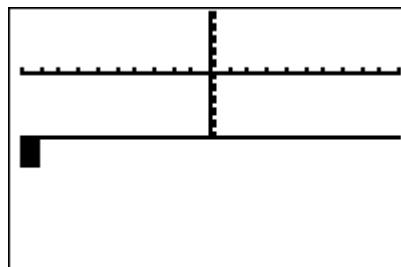
The Simul Command

Puts the calculator into simultaneous graphing mode. When multiple equations are enabled at the same time, sequential graphing mode means that they will be graphed at the same time (as opposed to Sequential mode, in which they will be graphed one after the other)

If you use a list in an equation, as with $Y1=\{1,2,3\}X$, this will graph several equations that will always graph separately, regardless of this setting, which only affects multiple functions in different equation variables.

Related Commands

- Sequential



Command Summary

Puts the calculator into simultaneous graphing mode.

Command Syntax

Simul

Menu Location

While editing a program, press:

1. MODE to access the mode menu.
2. Use arrows and ENTER to select Simul.

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/simul>

The $\sin^{-1}($ Command

$\sin^{-1}($ returns the arcsine of its argument. It is the inverse of $\sin($, which means that $\sin^{-1}(z)$ produces an angle θ such that $\sin(\theta)=z$.

$\sin^{-1}(.5$	5235987756
$\sin^{-1}(i$	

Like `sin()`, the result of `sin-1(` depends on whether the calculator is in Radian or Degree mode. However, unlike sine, the result is in degrees or radians, not the argument. A full rotation around a circle is 2π radians, which is equal to 360° . The conversion of $\theta = \sin^{-1}(n)$ from radians to degrees is $\theta * 180/\pi$ and from degrees to radians is $\theta * \pi/180$. The `sin-1(` command also works on lists.

The `sin-1(` function can be defined for all real and complex numbers; however, the function assumes real values only in the closed interval $[-1, 1]$. Because the trigonometric functions and their inverses in the Z80 calculators are restricted only to real values, the calculator will throw ERR:DOMAIN if the argument is outside of this interval, no matter what the mode setting may be.

In radians:

```
:sin-1(1)
1.570796327
```

In degrees:

```
:sin-1(1)
90
```

Advanced Uses

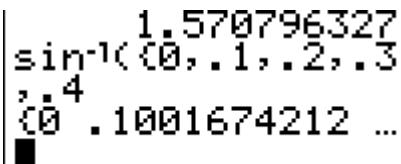
Since the function `sine` itself doesn't have the restrictions that `arcsine` does, and since `arcsine` is the inverse of `sine`, you can use `sin-1(sin(` to keep a variable within a certain range (most useful on the graph screen). Here is an example for a game like pong. The ball travels between -6 and 6.

You could use a flag like this:

```
:If 6=abs(X)           \ X is the position
:-D→D                 \ D is the direction
:X+D→X                \ new position
:Pt-On(-54,X,"=")
```

An easier way to do this, without needing a flag or even an If statement, is using `sin-1(sin(`

```
:X+1→X           \ Note: the calculator is in degree mode
:Pt-On(-54,sin-1(sin(15X))/15,"=")    \ 15 is used because sin-1 ranges
                                                and X from [-6, 6], so 90/
```



Command Summary

Returns the inverse sine (also called arcsine)

Command Syntax

`sin-1(number)`

Menu Location

Press:

1. [2nd]
2. [`sin-1`]

Calculator Compatibility

TI-83/84/+SE

Token Size

1 byte

Error Conditions

- **ERR:DATA TYPE** is thrown if you input a complex value or a matrix.
- **ERR:DOMAIN** is thrown if you supplied an argument outside the interval [-1,1]

Related Commands

- sin(
- cos(
- cos⁻¹(
- tan(
- tan⁻¹(

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/arcsin>

The sin(Command

NOTE: Due to the limitations of the wiki markup language, the L command on this page does not appear as it would on the calculator. See [Wiki Markup Limitations](#) for more information.

`sin(θ)` returns the sine of θ , which is defined as the y-value of the point of intersection of the unit circle and a line containing the origin that makes an angle θ with the positive x-axis

The value returned depends on whether the calculator is in Radian or Degree mode. A full rotation around a circle is 2π radians, which is equal to 360° . The conversion from radians to degrees is $\text{angle} * 180/\pi$ and from degrees to radians is $\text{angle} * \pi/180$. The `sin(` command also works on a list of real numbers.

In radians:

```
sin(π/6)
.5
```

In degrees:

```
sin(30)
.5
```

<code>sin(0</code>	<code>0</code>
<code>sin(180°</code>	<code>0</code>
<code>sin(π/6</code>	<code>.5</code>

Command Summary

Returns the sine of a real number.

Command Syntax

`sin(angle)`

Menu Location

Press the SIN key to paste `sin(`.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

Advanced Uses

You can bypass the mode setting by using the $^\circ$ (degree) and L (radian) symbols. These next

two commands will return the same values no matter if your calculator is in degrees or radians:

```
sin(30°)
.5
```

```
sin(π/6^r)
.5
```

Error Conditions

- **ERR:DATA TYPE** is thrown if you supply a matrix or a complex argument.

Related Commands

- [sin⁻¹\(](#)
- [cos\(](#)
- [cos⁻¹\(](#)
- [tan\(](#)
- [tan⁻¹\(](#)

See Also

- [Look-Up Tables](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/sin>

The sinh⁻¹(Command

The sinh⁻¹(command calculates the inverse hyperbolic sine of a value. sinh⁻¹(x) is the number y such that x = sinh(y). Unlike for the standard trig functions, this uniquely determines the inverse hyperbolic sine of any real number.

The sinh⁻¹(command also works for lists.

```
sinh¹(0)
0
sinh¹({1,2,3})
{ .881373587 1.443635475 1.818446
```

```
sinh¹(1     .881373587
sinh¹(10    2.99822295
sinh¹(E99   228.6490714
```

Command Summary

Calculates the inverse hyperbolic sine of a value.

Command Syntax

`sinh¹(value)`

Menu Location

The sinh⁻¹(command is only found in the catalog. Press:

Related Commands

- [sinh\(](#)
- [cosh\(](#)
- [cosh⁻¹\(](#)

- [tanh\(](#)
- [tanh⁻¹\(](#)

In the Catalog. Press:

1. 2nd CATALOG to access the command catalog.
2. S to skip to commands starting with S.
3. Scroll down and select sinh⁻¹(

Calculator Compatibility

TI-83/84/+SE

Token Size

1 byte

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/arsinh>

The sinh(Command

Calculates the hyperbolic sine of a value. The hyperbolic trig functions sinh(), cosh(), and tanh() are an analog of normal trig functions, but for a hyperbola, rather than a circle. They can be expressed in terms of real powers of e, and don't depend on the Degree or Radian mode setting.

```
sinh(0)
0
sinh(1)
1.175201194
```

Like normal trig commands, sinh() works on lists as well, but not on complex numbers, even though the function is often extended to the complex numbers in mathematics.

Formulas

The definition of hyperbolic sine is:

$$\sinh x = \frac{e^x - e^{-x}}{2} \quad (1)$$

Related Commands

- [sinh⁻¹\(](#)
- [cosh\(](#)
- [cosh⁻¹\(](#)

```
sinh(1
      1.175201194
sinh(230
      3.86100925e99
cosh(5)^2-sinh(5)
2
1
```

Command Summary

Calculates the hyperbolic sine of a value.

Command Syntax

sinh(value)

Menu Location

The sinh() command is only found in the Catalog. Press:

1. 2nd CATALOG to access the command catalog.
2. S to skip to commands starting with S.
3. Scroll down and select sinh().

Calculator Compatibility

TI-83/84/+SE

Token Size

- tanh(
- tanh⁻¹(

1 byte

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/sinh>

The SinReg Command

SinReg tries to fit a sine wave to a given list of points. To use it, you must first store the points to two lists: one of the x-coordinates and one of the y-coordinates, ordered so that the i th element of one list matches up with the i th element of the other list (i.e. the first element of the x-list and the first element of the y-list make up an ordered pair). L1 and L2 are the default lists used, and the List Editor (STAT > Edit...) is a useful window for entering the points.

SinReg requires that the lists contain at least 4 points. Also, if you do not provide two data points per cycle, the calculator may return a wrong answer. These conditions are an absolute minimum, and the command may fail to work even when they are met, and throw a ERR:SINGULAR MAT error. This is also likely to happen if the data are not actually periodic in nature.

In addition, to use SinReg in its simplest form, the x-coordinates must be sorted in increasing order, and the difference between consecutive x-coordinates must be the same throughout (i.e., $x_{i+1} - x_i$ should be the same for all i). You can then call SinReg with no arguments, and it will attempt to fit a sine wave to the data in L1 and L2:

```
: {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 → L1
: {21, 24, 34, 46, 58, 67, 72, 70, 61, 50, 40, 2
: SinReg
```

On the home screen, or as the last line of a program, this will display the equation of the curve: you'll be shown the format, $y=a\sin(bx+c)+d$, and the values of a , b , c and d . It will also be stored in the RegEQ variable, but you will not be able to use this variable in a program - accessing it just pastes the equation wherever your cursor was. Finally, the statistical variables a , b , c , and d will be set to the values computed as well. There are no correlation statistics available for SinReg even if Diagnostic Mode is turned on (see DiagnosticOn and DiagnosticOff).

A word of caution: the calculator assumes that Radian mode is enabled. If the calculator is set

```
SinReg
y=a*sin(bx+c)+d
a=1.415649772
b=1.858684022
c=-.7047761156
d=-.6236117962
```

Command Summary

Calculates the least-squares best fit sinusoidal curve through a set of points.

Command Syntax

SinReg [*iterations*, *x-list*, *y-list*, *period*, *equation*]

Menu Location

Press:

1. STAT to access the statistics menu
2. LEFT to access the CALC submenu
3. ALPHA C to select SinReg, or use arrows

Calculator Compatibility

TI-83/84/+SE

Token Size

1 byte

to Degree mode, the equation will still be in terms of radians: it will be correct, but values plugged in will give wrong answers. You will have to either switch to Radian mode, or multiply the values of b and c by $180/\pi$.

You do not have to do the regression on L1 and L2, in which case you'll have to enter the names of the lists after the command. For example:

```
:{1,2,3,4,5,6,7,8,9,10,11,12→MONTH  
:{21,24,34,46,58,67,72,70,61,50,40,27→TEMP  
:SinReg LMONTH,LTEMP
```

Unlike the other regression commands, SinReg does not allow you to use a frequency list for data. You can get around this by adding repeating coordinates multiple times.

The optional argument *iterations* should come before the data lists, and if provided will change the amount of time and effort the calculator spends on the problem. The value should be an integer 1 to 16; larger numbers mean greater precision, but a longer calculation time. The default value is 3, and for good reason: with a high precision value, the calculation may take a minute to complete, or longer, depending on the complexity of the problem.

The optional argument *period* should be given after the data lists - this is the length of a complete cycle in the data, if known. You might know the exact value of the period, for example, when the calculation involves time - a complete cycle could be a day, a month, or a year. Providing this argument is strongly recommended whenever it is available: this removes conditions on the x-coordinates' order and increment, and makes the calculation much faster and more accurate. If you have previously done a SinReg fit and desire a refined estimate, the value $2\pi/b$ can be given as the period.

Finally, you can enter an equation variable (such as Y_1) after the command, so that the curve's equation is stored to this variable automatically. This does not require you to supply the names of the lists, but if you do, the equation variable must come last. You can use polar, parametric, or sequential variables as well, but since the equation will be in terms of X anyway, this does not make much sense.

An example of SinReg with all the optional arguments:

```
:{1,2,3,4,5,6,7,8,9,10,11,12→MONTH  
:{21,24,34,46,58,67,72,70,61,50,40,27→TEMP  
:SinReg 16,LMONTH,LTEMP,12,Y1
```

The Levenberg-Marquardt nonlinear least-squares algorithm is used by SinReg.

Error Conditions

- ERR:SINGULAR MAT is thrown if you don't provide the calculator at least 4 points, or two data points per cycle.

Related Commands

- LinReg(ax+b)
- ExpReg
- Logistic

The solve(Command

The `solve(` command attempts to iteratively find a root of a given equation, given the variable to solve for, and an initial guess; i.e., given $f(x)$, `solve(` will attempt to find a value of x such that $f(x)=0$. `solve(` can take a list `{lower,upper}` as an optional fourth argument, in which case it attempts to find a root between *lower* and *upper* inclusive (by default, *lower* and *upper* are taken to be -E99 and E99 respectively). Brent's method is used for finding the root.

Unfortunately, the `solve(` command (as with most iterative methods) is not perfect at solving equations. `solve(` will in general be unable to find "multiple roots", or can only find it to an accuracy less than the usual (an example would be the root $x=1$ of the equation $(x-1)^n=0$ for n greater than 1). `solve(` will only return one of many possible roots to your equation if your equation has many roots to begin with. The root returned, in general, depends on the value of the guess given. The root returned is usually the root closest to the guess given for well-behaved equations; bad choices of the guess can cause `solve(` to either return a faraway root or not converge at all to a root.

If possible, the equation should first be solved by hand - if there is a relatively simple formula for the root, that will (usually) be more efficient than using `solve()`. Otherwise, ensure that the `solve(` call actually works in all the expected cases during use.

The Solver... utility (located in the same menu in the same place) is usually much easier and more intuitive to use, and is recommended instead of directly using `solve(` whenever applicable (e.g. the home screen). The same limitations apply to its efficiency. If you are unable to find roots using the Solver, try graphing the function and scanning for roots manually, then using 2:zero in the 2nd:CALC menu to refine your guess.

Note: Solver... changes the value of the variable being solved for to the root found; `solve(`, on the other hand, finds the root, but does not modify the original value of the variable.

Advanced Uses

Reformulating an equation may be useful in certain instances. For example, the equations

```
solve(X^2-5X,X,2,  
{-E99,E99} 0  
solve(X^2-5X,X,2,  
{1,E99} 5
```

Command Summary

Attempts to return a solution to the equation $expression=0$ for a specified single variable (other variables will be treated as constants), given a guess, and optionally bounds on the values of the variable.

Command Syntax

`solve(expression, variable, guess, [{lower, upper}])`

Menu Location

While editing a program, press:

1. MATH to go to the MATH menu.
2. 0 to choose `solve(`, or use arrows.

(outside the program editor, the interactive Solver will appear instead; use the Catalog to access the function directly)

Calculator Compatibility

TI-83/84/+SE

Token Size

1 byte

$f(x)=0$ and $e^{f(x)}=1$ are equivalent. `solve((X+1)^2,X,0` returns ERR:NO SIGN CHG, while `solve(e^((X+1)^2)-1,X,0` returns -1.000000616 (pretty close to the root -1). Rearranging the equation may sometimes help as well.

Specifying bounds usually helps `solve(` to find roots more efficiently. If bounds are readily available, they should be supplied to `solve(`.

Related Commands

- [fMax\(](#)
- [fMin\(](#)
- [fnInt\(](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/solve>

The SortA(Command

The `SortA(` command sorts a list in ascending order. It does not return it, but instead edits the original list variable (so it takes only list variables as arguments).

`SortA(` can also be passed multiple lists. In this case, it will sort the first list, and reorder the others so that elements which had the same indices initially will continue having the same indices. For example, suppose the X and Y coordinates of some points were stored in `LX` and `LY`, so that the I th point had coordinates `LX(I)` and `LY(I)`. Then `SortA(LX,LY)` would sort the points by their x-coordinates, still preserving the same points.

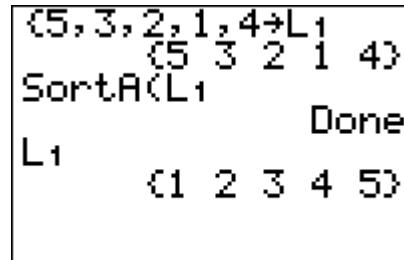
However, `SortA(` is not stable: if several elements in the first list are equal, then the corresponding elements in the subsequent lists may still end up being in a different order than they were initially.

Algorithm

The algorithm used by `SortA(` and `SortD(` appears to be a modified selection sort. It is still $O(n^2)$ on all inputs, but for some reason takes twice as long on a list with all equal elements. It is not stable.

Related Commands

- [SortD\(](#)



```
{5,3,2,1,4} → L1
{5 3 2 1 4}
SortA(L1)
Done
L1
{1 2 3 4 5}
```

Command Summary

Sorts a list in ascending order.

For more than one list, sorts the first, and reorders other lists accordingly.

Command Syntax

`SortA(list1 [,list2, ...])`

Menu Location

Press:

1. 2nd LIST to access the list menu.
2. RIGHT to access the OPS submenu.
3. ENTER to select `SortA(`.

Calculator Compatibility

TI-83/84+/SE

Token Size

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/sortd>

The SortD(Command

The SortD(command sorts a list in descending order. It does not return it, but instead edits the original list variable (so it takes only list variables as arguments).

SortD(can also be passed multiple lists. In this case, it will sort the first list, and reorder the others so that elements which had the same indices initially will continue having the same indices. For example, suppose the X and Y coordinates of some points were stored in LX and LY , so that the I th point had coordinates $\text{LX}(I)$ and $\text{LY}(I)$. Then SortD(LX, LY) would sort the points by their x-coordinates, still preserving the same points.

However, SortD(is not stable: if several elements in the first list are equal, then the corresponding elements in the subsequent lists may still end up being in a different order than they were initially.

Algorithm

The algorithm used by SortD(and SortA(appears to be a modified selection sort. It is still $O(n^2)$ on all inputs, but for some reason takes twice as long on a list with all equal elements. It is not stable.

Related Commands

- [SortA\(](#)

```
(5,3,2,1,4→L1
      (5 3 2 1 4)
SortD(L1
      Done
L1
      (5 4 3 2 1)
```

Command Summary

Sorts a list in descending order.

For more than one list, sorts the first, and reorders other lists accordingly.

Command Syntax

`SortD(list1 [,list2, ...])`

Menu Location

Press:

1. 2nd LIST to access the list menu.
2. RIGHT to access the OPS submenu.
3. 2 to select SortD(, or use arrows and ENTER.

Calculator Compatibility

TI-83/84+/SE

Token Size

1 byte

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/sortd>

The startTmr Command

The `startTmr` command is used with the built-in timer that is available on the TI-84+/SE calculators. It is used together with the `checkTmr(` command to determine how much time has elapsed since the timer was started. An application of these commands is timing different commands or pieces of code, as well as countdowns in games, or a time-based score (such as in Minesweeper).

To use the timer, you first store `startTmr` to a variable (usually, a real variable) whenever you want the count to start. Now, whenever you want to check the elapsed time, you can use `checkTmr(` with the variable from above, giving you the number of seconds that have passed. Using `checkTmr(` doesn't stop the timer, you can do it as many times as you want to.

In the case of Minesweeper, for example, you would store `startTmr` to, for example, `T`, after setting up and displaying the board, display the result of `checkTmr(T)` in the game's key-reading loop, and store `checkTmr(T)` to the player's score if he wins.

Despite the name of the command, `startTmr` doesn't start the clock if it's stopped; use `ClockOn` instead to start the clock.

Advanced Uses

To time a command or routine using `startTmr` and `checkTmr(`, use the following template:

```
:ClockOn  
:startTmr→T  
:For(A,1,(number)  
    (command(s) to be tested)  
:End  
:checkTmr(T)/(number)
```

Making (number) higher increases accuracy, but takes longer. Also, make sure not to modify the variables A or T inside the `For(` loop.

While this method eliminates human error from counting, it's prone to its own faults. A major one is that `startTmr` and `checkTmr(` always return whole numbers, but time is continuous. Depending on how close the start and end of the loop were to a clock tick, the number of seconds may be off by up to one second in either direction. To take this into account, you could replace the last line:

```
: (checkTmr(T)+{-1,1})/(number)
```

This will give you a list of the maximum and minimum possible times — the true time that the

```
PROGRAM: TIMER  
:startTmr→A  
:For(B,1,randInt  
( $\leq$ 2,5000  
:End  
:Disp checkTmr(A  
■
```

Command Summary

Returns the value of the clock timer on the TI-84+/SE.

Command Syntax

`startTmr→Variable`

Menu Location

This command can only be found in the catalog. Press:

1. 2nd CATALOG to enter the command catalog
2. T to skip to command starting with T
3. Scroll up to `startTmr` and select it

Calculator Compatibility

TI-84+/SE

Token Size

2 bytes

command takes is guaranteed to be somewhere in between.

The other thing you need to be aware of when testing code is that there are many different things that will affect the time: the strength of the batteries, the amount of free RAM, and including the closing parenthesis on the For(loop. The last one, in particular, has an impact when using a lone If command or one of the IS>(or DS<(commands.

Related Commands

- checkTmr(

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/starttmr>

The stdDev(Command

The stdDev(command finds the sample standard deviation of a list, a measure of the spread of a distribution. It takes a list of real numbers as a parameter. For example:

```
:Prompt L1  
:Disp "STD. DEV. OF L1",stdDev(L1)
```

Caution: the standard deviation found by this command is the **sample** standard deviation, not the **population** standard deviation, which is the one most commonly used when dealing with a sample rather than the whole population. The formula for population standard deviation is similar, but $N-1$ is replaced by N . There is no single command that will calculate population standard deviation for you, but 1-Var Stats will return both (sample standard deviation, the one returned by stdDev(), is S_x , while population standard deviation is σ_x). You can also calculate population standard deviation of L1 with the following code:

```
:stdDev(augment(L1,{mean(L1
```

Advanced Uses

Frequency lists don't need to be whole numbers. Amazing as that may sound, your calculator can handle being told that one element of the list occurs $1/3$ of a time, and another occurs 22.7 times. It can even handle a frequency of 0 - it will just ignore that element, as though it weren't there. One caveat, though - if all of the elements occur 0 times, there's no elements actually in the list and your calculator will throw an error.

```
stdDev({1,2,3  
1  
stdDev({1,2,3},{  
1,0,5  
.8164965809
```

Command Summary

Finds the sample standard deviation of a list.

Command Syntax

`stdDev(list,[freqlist])`

Menu Location

Press:

1. 2ND LIST to enter the LIST menu.
2. LEFT to enter the MATH submenu.
3. 7 to select stdDev(, or use arrows.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

Formulas

The formula for standard deviation used by this command is:

$$s = \sqrt{\frac{1}{N-1} \sum_{i=1}^N (x_i - \bar{x})^2} \quad (1)$$

This is the formula for sample standard deviation. The formula for population standard deviation, which this command does **not** use, varies slightly:

$$\sigma = \sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - \bar{x})^2} \quad (2)$$

Related Commands

- [mean\(](#)
- [median\(](#)
- [variance\(](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/stddev>

The Stop Command

When the Stop command is used in a program it exits the program (terminating the program execution) and returns you to the home screen. If it is encountered within loops, the loops will be stopped.

There is some distinction when using Stop with subprograms: the Stop command will stop the program execution of the subprogram, as well as the calling program, and return you to the home screen; the program will stop completely. If this functionality is not desired, then you should use the Return command instead.

Optimization

You don't have to put a Stop command at the end of a program or subprogram if you can organize the program so that it just naturally quits. When the calculator reaches the end of a program, it will automatically stop executing as if it had encountered a Stop command (the Stop is implied).

PROGRAM: EXAMPLE
:Disp "THIS GETS
DONE
:Stop
:Disp "BUT THIS
DOESN'T

Command Summary

Completely stops the current program and any parent programs.

Command Syntax

Stop

Menu Location

While editing a program, press

1. PRGM to enter the program menu
2. ALPHA F to choose Stop, or

```
:ClrHome
:Input "Guess:",A
:Stop
Remove the Stop
:ClrHome
:Input "Guess:",A
```

use arrows

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

Error Conditions

- ERR:INVALID occurs if this statement is used outside a program.

Related Commands

- prgm
- Return

See Also

- Subprograms

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/stop>

The StoreGDB Command

The StoreGDB command stores all graph settings needed to reconstruct the current graph to a GDB (Graph DataBase) variable, one of GDB1, GDB2, ..., GDB0 (as indicated by the argument). These settings can then be recalled using the RecallGDB command.

The settings stored in a GDB include:

- The graphing mode currently enabled.
- All equations in the current graphing mode, but NOT other graphing modes.
- All window variables applicable to the current graphing mode. This does not include zoom variables, table settings, or irrelevant variables such as Tmin when in function mode.
- The menu settings relevant to graphing (everything in the 2nd FORMAT menu, as well as Connected/Dot and Sequential/Simul settings in the MODE menu)

The number passed to StoreGDB must be one of 0 through 9. It has to be a number: StoreGDB X will not work, even if X contains a value 0 through 9.

Advanced Uses

StoreGDB 1	Done
AxesOff	Done
RecallGDB 1	Done
	Done

Command Summary

Stores graph setting to a GDB (Graph DataBase) to be recalled later with RecallGDB.

Command Syntax

StoreGDB *number*

Menu Location

Press:

1. 2nd DRAW to access the draw menu.
2. LEFT to access the STO menu.
- 3 to select StoreGDB. or use

The StoreGDB and RecallGDB variables are useful in Cleaning up after a program finishes running, preserving the user's settings. If your program heavily relies on the graph screen, it may end up editing window size or other graph settings, which the user might want to be saved. This is easily done:

Add StoreGDB 1 (or any other number) to the beginning of your program.

Then, feel free to edit any graph settings you like.

At the end of your program, add RecallGDB 1, followed by DelVar GDB1, to recall the graph settings stored at the beginning.

GDBs can also be useful in adding extra string storage. You can store strings to the Yn variables, and back them up in a GDB; to retrieve them later, recall the GDB and use Equ▶String(to store the equations to the strings again.

Error Conditions

- ERR:DATA TYPE is thrown if the argument is not a *number* 0 through 9.

Related Commands

- RecallGDB

See Also

- Cleaning up

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/storegdb>

The StorePic Command

StorePic saves the graph screen to a picture (to recall it later, use RecallPic). Every detail of the graph screen will be stored as it appears, with the sole exception of X and Y labels on the axes (if they are shown).

The number passed to StorePic must be one of 0 through 9. It has to be a number: StorePic X will not work, even if X contains a value 0 through 9.

Advanced Uses

A combination of StorePic and RecallPic can be used to maintain a background over which another sprite moves:

First, draw the background, and save it to a picture

arrows and ENTER.

Calculator Compatibility

TI-83/84/+SE

Token Size

1 byte



Command Summary

Stores the graph screen to a picture (one of Pic1, Pic2, ..., Pic0)

Command Syntax

StorePic *number*

file with StorePic.

Next, draw the sprite to the screen.

When you want to move the sprite, erase it, then use RecallPic to draw the background again.

Then draw the sprite to its new location on the screen again (this can be done before or after using RecallPic).

Also, if a screen in your program takes more than a second to draw, and is displayed several times, you might want to consider storing it to a picture the first time it's drawn, and then recalling it every next time you want to draw it.

Error Conditions

- **ERR:DATA TYPE** is thrown if the argument is not a **number** 0 through 9.

Related Commands

- [ClrDraw](#)
- [RecallPic](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/storepic>

The String►Equ(Command

This command stores the contents of a string to an equation variable (such as Y_1 or X_{1T}). This command can, in theory, be used whenever you need to set any equation variable.

In practice, however, this command is useless. This is because the \rightarrow (store) operation can be used for the same purpose:

```
:String►Equ(Str1, Y1  
can be  
:Str1→Y1
```

This replacement is universal, takes the same time to run (because it actually uses the same routines), is more convenient to type since you don't have to go through the command catalog, and is several bytes smaller.

Advanced

Unlike any normal use of the \rightarrow (store) operation,

Menu Location

Press:

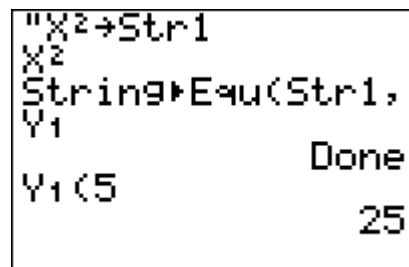
1. 2nd DRAW to access the draw menu.
2. LEFT to access the STO submenu.
3. ENTER to select StorePic

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte



Command Summary

Stores the contents of a string to an equation variable.

Command Syntax

`String►Equ(string, equation)`

Menu Location

This command is found only in the catalog. Press:

1. 2nd CATALOG to access the catalog

this situation is different because it doesn't modify Ans. For example:

```
:125  
:"sin(X→Y1  
:Disp Ans
```

Because this use of → does not modify Ans, '125' will be displayed rather than 'sin(X)'. However, if we were to replace Y1 with Str1, then the → operation would work normally, and 'sin(X)' would be displayed.

It's also important to realize the difference between the String►Equ(command and the related Equ►String(. I mean, aside from the fact that the latter actually is useful. The main difference is that while Equ►String('s arguments both have to be **variables**, String►Equ('s first argument can either be a variable (Str0 through Str9), a constant string (e.g., "sin(X)'), or an expression that returns a string (e.g., sub(Str1,1,5)). This applies to the → operation as well.

Related Commands

- Equ►String(
- expr(

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/string-equ>

The sub(Command

The sub(command is used to get a substring, or a specific part of a string. It takes three arguments: *string* is the source string to get the substring from, *start* is the index of the token to start from, and *length* is the length of the substring you want. For example:

```
:sub("TI-BASIC",4,5  
"BASIC"  
:sub("TI-BASIC",5,2  
"AS"
```

Keep in mind that you can't have an empty string, so the *length* argument can't be equal to 0.

When the *length* argument is 1, sub(*string*,N,1 returns the Nth token in the string.

Advanced Uses

If only one argument is given, and it contains an expression that evaluates to a real or complex

2. T to skip to commands starting with T
3. Scroll up to String►Equ(and select it.

Calculator Compatibility

TI-83/84/+SE

Token Size

2 bytes

```
sub("CAT DOG",5,  
3  
DOG
```

Command Summary

Returns a specific part of a given string, or divides by 100.

Command Syntax

sub(*string*, *start*, *length*)

sub(*value*)

Menu Location

This command can only be found in the catalog. Press:

number or list of numbers, the sub(command will divide the result by 100.

```
:sub(225  
     2.25  
:sub({3+5i,-4i►Frac  
     {3/100+1/20i,-1/25i}
```

Much like the use of the % symbol, this is an undocumented feature that was introduced in OS version 1.15. Thus, care should be taken when using sub(in this way, as older versions will not support it.

Together with the inString(command, sub(can be used to store a "list of strings" in a string, that you can then get each individual string from. To do this, think of a delimiter, such as a comma, to separate each individual string in the "list" (the delimiter must never occur in an individual string). The code will be simpler if the delimiter also occurs at the end of the string, as in "CAT,DOG,RAT,FISH,".

This routine will display each string in a "list of strings". You can adapt it to your own needs.

```
:1→I  
:inString(Str1,",",→J  
:While Ans  
:Disp sub(Str1,I,J-I  
:J+1→I  
:inString(Str1,",",Ans→J  
:End
```

This routine allows created random letters instead of random numbers.

```
Repeat Getkey  
Disp sub("ABCDEFGHIJKLMNPQRSTUVWXYZ", randint(1,26), 1  
End
```

Related Commands

- %
- expr(
- inString(
- length(

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/sub>

1. 2nd CATALOG to enter the command catalog
2. T to skip to command starting with T
3. Scroll up to sub(and select it

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

The sum(Command

The sum(command calculates the sum of all or part

of a list.

When you use it with only one argument, the list, it sums up all the elements of the list. You can also give it a bound of *start* and *end* and it will only sum up the elements starting and ending at those indices (inclusive).

```
sum({1,2,3,4,5})  
15  
sum({1,2,3,4,5},2,4)  
9  
sum({1,2,3,4,5},3)  
12
```

Optimization

If the value of *end* is the last element of the list, it can be omitted:

```
sum({1,2,3,4,5},3,5)  
can be  
sum({1,2,3,4,5},3)
```

Error Conditions

- **ERR:DOMAIN** is thrown if the starting or ending value aren't positive integers.
- **ERR:INVALID DIM** is thrown if the starting or ending value exceed the size of the list, or are in the wrong order.

Related Commands

- [prod\(](#)
- [dim\(](#)
- [seq\(](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/sum>

The \tan^{-1} (Command

$\tan^{-1}($ returns the arctangent of its argument. It is the inverse of $\tan($, which means that $\tan^{-1}(n)$ produces an angle θ such that $\tan(\theta)=n$.

Like $\tan($, the result of $\tan^{-1}($ depends on whether the calculator is in Radian or Degree mode. However, unlike tangent, the result is in degrees or

{5,4,3,2,1}→L ₁	
5 4 3 2 1	
sum(L ₁)	15
sum(L ₁ ,2,4)	9

Command Summary

Calculates the sum of all or part of a list.

Command Syntax

`sum([list,[start],[end]])`

Menu Location

Press:

1. 2nd LIST to access the list menu.
2. LEFT to access the MATH submenu.
3. 5 to select sum(, or use arrows and ENTER.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

$\tan^{-1}(0)$	0
$\tan^{-1}(-1,0,1)$	0
$\tan^{-1}(-.7853981634)$	0...
$2\tan^{-1}(e^{99})$	
	3.141592654

radians, not the argument. A full rotation around a circle is 2π radians, which is equal to 360° . The conversion of $\theta = \tan^{-1}(n)$ from radians to degrees is $\theta * 180/\pi$ and from degrees to radians is $\theta * \pi/180$. The \tan^{-1} command also works on a list.

\tan^{-1} will always return a value between $-\pi$ and π (or -180° and 180°).

In radians:

```
:tan-1(1)  
.7853981634
```

In degrees:

```
:tan-1(1)  
45
```

Optimization

Expressions of the form $\tan^{-1}(y/x)$ are usually better recast as $R\blacktriangleright P\theta(x,y)$; the latter will not fail even if x should happen to be equal to zero.

Error Conditions

- **ERR:DATA TYPE** is thrown if you input a complex value or a matrix.

Related Commands

- [sin\(](#)
- [sin⁻¹\(](#)
- [cos\(](#)
- [cos⁻¹\(](#)
- [tan\(](#)
- [R \$\blacktriangleright\$ P \$\theta\(\$](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/arctan>

The tan(Command

$\tan(\theta)$ calculates the tangent of the angle θ , which is defined by $\tan \theta = \frac{\sin \theta}{\cos \theta}$

The value returned depends on whether the calculator is in Radian or Degree mode. A full rotation around a circle is 2π radians, which is equal to 360° . The conversion from radians to degrees is

Command Summary
Returns the inverse tangent (also called arctangent)
Command Syntax
$\tan^{-1}(number)$
Menu Location
Press:
1. [2nd] 2. [\tan^{-1}]
Calculator Compatibility
TI-83/84/+/SE
Token Size
1 byte

$\tan(0)$	0
$\tan(1.57)$	1255.765591
$\tan(45^\circ)$	1

$\text{angle} * 180/\pi$ and from degrees to radians is $\text{angle} * \pi/180$. The `tan(` command also works on a list of real numbers.

Since tangent is defined as the quotient of sine divided by cosine, it is undefined for any angle such that $\cos(\theta)=0$

In radians:

```
tan(π/4)  
1
```

In degrees:

```
tan(45)  
1
```

Advanced Uses

You can bypass the mode setting by using the $^\circ$ (degree) and r (radian) symbols. These next two commands will return the same values no matter if your calculator is in degrees or radians:

```
tan(45°)  
1
```

```
tan(π/4^r)  
1
```

Error Conditions

- **ERR:DATA TYPE** is thrown if you supply a matrix or a complex argument.
- **ERR:DOMAIN** is thrown if you supply an angle of $\pi/2 \pm n\pi$ (in radians, where n is an integer) or $90 \pm 180n$ (in degrees, where n is an integer)

Related Commands

- [sin\(](#)
- [sin⁻¹\(](#)
- [cos\(](#)
- [cos⁻¹\(](#)
- [tan⁻¹\(](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/tan>

Command Summary

Returns the tangent of a real number.

Command Syntax

`tan(angle)`

Menu Location

Press TAN

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

The Tangent(Command

The `Tangent(` command draws a graph of an expression and then draws a line tangent to that expression, with the line touching the graph at the point of the specified value. You can either use a equation variable (such as Y_1) or an expression in terms of X (such as X^2). Though you can use equation variables from any graphing mode, they will be treated as functions in terms of X. `Tangent(` also ignores the graphing mode currently selected.

Here is a simple example, where we are graphing the parabola X^2 and then drawing a tangent line at the value $X=2$.

```
: "X2→Y1
:Tangent(Y1, 2
```

or

```
: Tangent(X2, 2
```

Advanced Uses

Whether the graph shows up or not is dependent on the window dimensions of the graph screen, and you should use a friendly window to ensure it shows up as you intended.

`Tangent(` will update X and Y for each coordinate drawn (like `DrawF` and `DrawInv`), and exit with the last coordinate still stored.

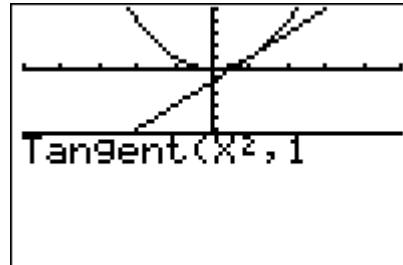
When evaluating the expression using `Tangent(`, the calculator will ignore the following errors: ERR:DATA TYPE, ERR:DIVIDE BY 0, ERR:DOMAIN, ERR:INCREMENT, ERR:NONREAL ANS, ERR:OVERFLOW, and ERR:SINGULAR MAT. If one of these errors occurs, the data point will be omitted. However, the errors will still be thrown if they occur when evaluating the function *at* the point of tangency.

Using Ans as an optimization for storing to an equation will not work. For example, the following code returns ERR:DATA TYPE because Ans is a string, not an equation variable.

```
: "X2
:Tangent(Ans, 2
```

Of course, you *can* use Ans in the equation, if it's a real number, but that's usually not as useful.

Error Conditions



Command Summary

Draws a line tangent to an expression at the specified value.

Command Syntax

`Tangent(expression,value)`

Menu Location

Press:

1. 2nd PRGM to access the draw menu.
2. 5 to select `Tangent(`, or use arrows and ENTER.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

- [ERR:INVALID](#) is thrown if you try to use an equation variable that is undefined.

Related Commands

- [DrawF](#)
- [DrawInv](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/tangent>

The $\tanh^{-1}($ Command

The $\tanh^{-1}($ command calculates the inverse hyperbolic tangent of a value. $\tanh^{-1}(x)$ is the number y such that $x = \tanh(y)$.

$\tanh^{-1}(x)$, although it can be defined for all real and complex numbers, is real-valued only for x in the open interval $(-1,1)$. Since Z80 calculators have their hyperbolic functions and inverses restricted to real values, [ERR:DOMAIN](#) is thrown when x is outside the interval $(-1,1)$.

The $\tanh^{-1}($ command also works for lists.

```
tanh-1(0)
0
tanh-1({- .5, .5})
{ - .5493061443 .5493061443 }
```

$\tanh^{-1}(.5)$	$.5493061443$
$\tanh^{-1}(.75)$	$.9729550745$
$\tanh^{-1}(.9999)$	4.951718776

Command Summary

Calculates the inverse hyperbolic tangent of a value.

Command Syntax

$\tanh^{-1}(\textit{value})$

Menu Location

The $\tanh^{-1}($ command is only found in the catalog. Press:

1. 2nd CATALOG to access the command catalog.
2. T to skip to commands starting with T.
3. Scroll down and select $\tanh^{-1}($

Calculator Compatibility

TI-83/84/+SE

Token Size

1 byte

Error Conditions

- [ERR:DOMAIN](#) when taking the inverse tanh of a number not between -1 and 1.

Related Commands

- [sinh\(](#)
- [sinh⁻¹\(](#)
- [cosh\(](#)
- [cosh⁻¹\(](#)
- [tanh\(](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/artanh>

The tanh(Command

Calculates the hyperbolic tangent of a value. The hyperbolic trig functions sinh(, cosh(, and tanh(are an analog of normal trig functions, but for a hyperbola, rather than a circle. They can be expressed in terms of real powers of e, and don't depend on the Degree or Radian mode setting.

```
tanh(0)  
0  
tanh(1)  
.761594156
```

Like normal trig commands, tanh(works on lists as well, but not on complex numbers, even though the function is often extended to the complex numbers in mathematics.

Advanced Uses

The tanh(command can be used to approximate the sign function:

$$x = \begin{cases} -1 & \text{if } x < 0, \\ \text{if } x = 0, & \text{if } x > 0. \end{cases} \quad (1)$$

As the absolute value of the input becomes large, the convergence is achieved at a point closer to zero. For the function to work as intended generally, numbers having lesser orders of magnitude need to be multiplied by a factor large enough for the argument to arrive at ± 16.720082053122 , which is the smallest input to produce ± 1 (respectively) to fourteen digits of accuracy.

```
5/12→X  
.4166666667  
tanh(E9X)  
1  
tanh(-E9X)  
-1
```

```
tanh(1  
.761594156  
tanh((1,2,3  
(.761594156 .96...  
tanh(E99  
1
```

Command Summary

Calculates the hyperbolic tangent of a value.

Command Syntax

tanh(value)

Menu Location

The tanh(command is only found in the Catalog. Press:

1. 2nd CATALOG to access the command catalog.
2. T to skip to commands starting with T.
3. Scroll down and select tanh(.

Calculator Compatibility

TI-83/84/+SE

Token Size

1 byte

Formulas

The definition of the hyperbolic tangent is:

$$\tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1}$$

Related Commands

- [sinh\(](#)
- [sinh⁻¹\(](#)
- [cosh\(](#)
- [cosh⁻¹\(](#)
- [tanh⁻¹\(](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/tanh>

The tcdf(Command

tcdf(is the Student's *t* cumulative density function. If some random variable follows this distribution, you can use this command to find the probability that this variable will fall in the interval you supply.

Unlike [normalcdf\(](#), this command only works for the standardized distribution: with mean 0 and standard deviation 1. To use it for non-standardized values you will have to standardize them by calculating $(X - \mu)/s$ (where μ is the mean, and s the standard deviation). Do this for both *lower* and *upper*.

Example of using tcdf(:

```
for the probability of being within
:tcdf(-2,2,5
```

```
tcdf(-1,1,15
      .6668298639
tcdf(-1,1,45
      .6773423474
tcdf(-1,1,1000
      .6824475815
```

Command Summary

Calculates the Student's *t* probability between *lower* and *upper* for specified degrees of freedom.

Command Syntax

tcdf(*lower*, *upper*, *df*)

Menu Location

Press:

1. 2ND DISTR to access the distribution menu
2. 5 to select tcdf(, or use arrows.

Press 6 instead of 5 on a TI-84+/SE with OS 2.30 or higher.

Calculator Compatibility

TI-83/84+/SE

Token Size

2 bytes

Advanced

Often, you want to find a "tail probability" - a special case for which the interval has no lower or no upper bound. For example, "what is the probability x is greater than 2?". The TI-83+ has no special symbol for infinity, but you can use E99 to get a very large number that will work equally well in this case (E is the decimal exponent obtained by pressing [2nd] [EE]). Use E99 for positive infinity, and -E99 for negative infinity.

Formulas

As with any other continuous distribution, tcdf(can be defined in terms of the probability density function, [tpdf\(](#):

$$\text{tcdf}(a, b, k) = \int_a^b \text{tpdf}(t, k) dt \quad (1)$$

The function can also be expressed in terms of an [incomplete beta function](#).

For the special case df=1, tcdf(is expressible in terms of simpler functions:

$$\text{tcdf}(a, b, 1) = \frac{1}{2} + \frac{\tan^{-1}(x)}{\pi} \quad (2)$$

This is the so-called Cauchy distribution.

Related Commands

- [tpdf\(\)](#)
- [invT\(\)](#)
- [Shade_t\(\)](#)

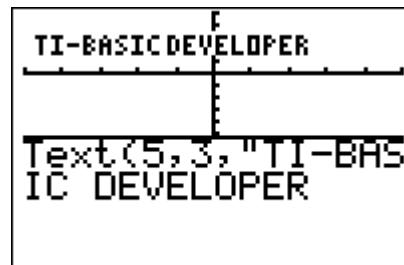
For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/tcdf>

The Text(Command

The Text(command allows you to display text on the graph screen, using the small font. It takes three arguments: the row (0-62) at which you want to display something, the column (0-94), and whatever it is you want to display. Like the [Output\(](#) command, it is limited to numbers and strings. If part of what you want to display goes off the screen, it will not be displayed - the calculator will cut you off at the most characters that will fit on the screen entirely.

Unlike the large text used on the home screen, the small font this command uses varies in width from 2 pixels to as many as 6 (counting the blank space at the end of each character, which is 1 pixel). All characters are 6 pixels tall, but the top row of pixels is used very rarely (only in international characters such as ä). On the TI-84+ and TI-84+ SE, the Text(command may also erase a single row of pixels underneath the text: whether this occurs or not depends on whether it was the menu screen or the table that was visited last, of the two.

Without storing them to a special string, the Text(command cannot be used to display quotation marks ("") and the [store](#) command (\rightarrow). However, you can mimic these respectively by using two apostrophes (''), and two subtract signs and a greater than sign ($\rightarrow>$).



Command Summary

Displays a number, a string, or several numbers and strings (you may mix and match) at (row, column) on the graph screen, using the small font.

The alternative syntax allows using the large font on the graph screen (TI-83+ and higher only)

Command Syntax

`Text(row, column, value1 [,value2, ...])`

(83+ and higher only)

`Text(-1, row, column, value1 [,value2, ...])`

Like many other drawing commands, if you're outside a program and on the graph screen, you can use this command directly, without going to the home screen. Just select `Text(` from the draw menu, and you will be able to type text at a cursor you control with arrow keys; press `CLEAR` or `ENTER` (among other things) to exit this mode.

Advanced Uses

On the TI-83+, 83+SE, 84+, and 84+SE, `Text(` has an alternate syntax: put a `-1` before the row and column to display the text using the large font instead of the small font. With this syntax, `Text(` becomes like an `Output(` for the graph screen, but with more features: you don't have to display text exactly aligned to one of the home screen's rows and columns, and you can display more than one string or number at a time. Also, text still won't wrap like `Output('s)` does.

This feature may be helpful in making programs more appealing, but remember that it does not work on the regular TI-83. If you want to maintain compatibility, don't use this syntax, or make an alternate version of your program without it.

The `Text(` command is also critical to the sprite technique known as text sprites. Although they have limitations, they allow pure Basic programs to have high-quality graphics without taking up lots of space.

On the TI-84+ and TI-84+ SE, another compatibility issue occurs with `Text(`. On certain occasions, using `Text(` to display small text on the graph screen will erase a 1-pixel margin below the text itself. The cause is a system option which is turned on when accessing the new MODE menu, and turned off when accessing the table, matrix editor, or list editor. The 1-pixel margin may not seem like a big deal, but it's enough to stop certain games (such as Bryan Thomas's Contra) from working on the TI-84+/SE.

The situation can be detected quite easily: turn on a pixel, display text 6 rows above it, and test if the pixel is still turned on. Fixing the situation is slightly more difficult:

- The hex code AsmPrgmFD**C**B058EC9 will disable the option (but it requires having an additional subprogram).
- DispTable will also do the trick, but of course it will display the table as well.
- There's the option of telling users to access a certain screen before playing...

You can also try to get around the situation by storing and recalling pictures, to prevent anything from being erased when you don't want it to be.

Error Conditions

- **ERR:DOMAIN** is thrown if the coordinates of `Text(` are out of range: usually the range is between 0 and 57 for the row, and between 0 and 94 for the column. A few comments:
 - **ERR:DATA TYPE** can sometimes occur instead on the TI-83+ or higher because of confusion with the alternate syntax
 - Similarly, `Text(-1,0,0)` will cause no error and display nothing whatsoever on the TI-83+ or higher.
 - With `Text(-1,...` the upper bound on the row is one less of what it would be normally.

Menu Location

Press:

1. 2ND PRGM to enter the DRAW menu
2. 0 to choose `Text(`, or use arrows

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

- In Horiz mode the upper bound on the row is 25 rather than 57. In G-T mode the upper bound on the column is 46.
- **ERR:ARGUMENT** is thrown if the number of arguments given to Text(is 256 or more.

Related Commands

- Disp
- Output
- Pause

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/text>

The timeCnv(Command

The timeCnv(command converts seconds into the equivalent days, hours, minutes, and seconds. You just specify a number of seconds (should be a whole number, although a decimal number would work too; the calculator will simply use the integer part and discard the decimal) and the calculator will automatically break the seconds up into the standard parts of time, storing them in list format — {days, hours, minutes, seconds}. You can store this list to a variable for later use, or manipulate it the same way you do with other lists.

The number of seconds you specify can be as small or large as you want, although the number must be at least zero (in which case, the time list will be all zeroes). At the same time, you will run into the standard number precision problems that plague TI-Basic when specifying an extremely large or small number. Because of this, you should try to use numbers with less than 10 digits. Here is a simple example, where the time is exactly 1 day, 1 hour, 1 minute, and 1 second:

```
: timeCnv(90061→L1
: Disp Ans
```

The time conversion is 60 seconds for a minute, 3600 (60×60) seconds for an hour, and 86400 ($60 \times 60 \times 60$) seconds for a day. Adding these three together plus the one second gives you the value that you see in the example. This is pretty basic math, so it should be easy to understand.

Related Commands

- getTime
- getDate

```
timeCnv(4326543
        {50 1 49 3)
timeCnv(2479
        {0 0 41 19)
timeCnv(2636432
        {30 12 20 32)
```

Command Summary

Converts seconds into the equivalent days, hours, minutes, and seconds.

Command Syntax

timeCnv(*value*)→*variable*

Menu Location

This command can only be found in the catalog. Press:

1. 2nd CATALOG to enter the command catalog
2. t to skip to commands starting with T
3. Scroll down to timeCnv(and select it

Calculator Compatibility

TI-84+/SE

Token Size

2 bytes

The Time Command

NOTE: This article is about the Time setting for sequence graphing. If you're looking for the clock commands on the TI-84 Plus and TI-84 Plus SE, see [Time and Date Commands](#).

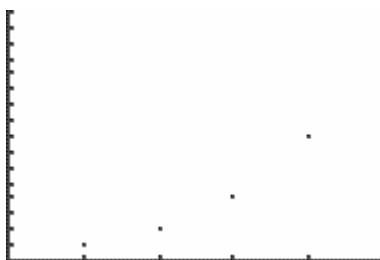
The Time command sets equations in sequence mode to graph as the points $(n, u(n))$ (for the u equation; $(n, v(n))$ and $(n, w(n))$ for the other two) - the default setting. In dot mode, only the points themselves will be plotted, but if you change the graphing style to connected line or thick line, the points will be connected.

Essentially, this mode makes sequence graphs a limited version of function graphs, but with the possibility of recursion.

See "Related Commands" for other possibilities of graphing sequences.

Related Commands

- [Web](#)
- [uvAxes](#)
- [uwAxes](#)
- [vwAxes](#)



Command Summary

Sets the way sequence equations are graphed to value vs. time.

Command Syntax

Time

Menu Location

While in Seq mode, press:

1. 2nd FORMAT to access the format menu.
2. ENTER to select Time

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

The TInterval Command

The TInterval command calculates a confidence interval for the mean value of a population, at a specific confidence level: for example, if the confidence level is 95%, you are 95% certain that the mean lies within the interval you get. Use TInterval when you have a single variable to analyze, and **don't** know the standard deviation. The TInterval assumes that your distribution is

```
TInterval  
(47.611,50.389)  
x̄=49  
Sx=.7  
n=100
```

normal, but it will work for other distributions if the sample size is large enough.

There are two ways to call the `TInterval` command: by supplying it with needed sample statistics (mean, sample standard deviation, and sample size), or by entering a list and letting the calculator work the statistics out.

Sample Problem

You want to know the average height of a student at your school. You haven't asked everyone, but you took a random sample of 30 people and found out their heights (and stored it to L_1). You've decided to use a 95% confidence interval.

Since the syntax for entering a data list is `TInterval list, confidence level`, here is your code:

```
:TInterval L1,95  
you can also use  
:TInterval L1,.95
```

Alternatively, you could calculate the mean, sample size, and standard deviation, and enter those instead. The sample size in this case is 30; let's say the mean was 63 inches and the standard deviation 6.2 inches.

The syntax for entering statistics is `TInterval mean, std. deviation, sample size, confidence level`, so your code would look like:

```
:TInterval 63,6.2,30,95  
you can also use  
:TInterval 63,6.2,30,.95
```

Of course, the main use of the `TInterval` command is in a program. While you can enter the `TInterval` command on the home screen as well (just look in the catalog for it), it would probably be easier to select `TInterval...` from the `STAT>TEST` menu (see the sidebar).

Advanced Uses

As with most other statistical commands, you can enter a second list after the data list, to add frequencies (only with the data list syntax, of course). The frequency list must contain non-negative integers, and can't be all 0.

Optimization

Using the data list syntax, all the arguments are optional: the calculator will assume you want to use L_1 for your data unless another list is supplied, and that the confidence level you want is

Command Summary

Using either already-calculated statistics, or a data set, computes a t confidence interval.

Command Syntax

`TInterval [//frequency//] [level//]
(data list input)`

`TInterval mean, std. deviation,
sample size, [level//]
(summary stats input)`

Menu Location

When editing a program, press:

1. STAT to access the statistics menu
2. LEFT to access the TESTS submenu
3. 8 to select `TInterval`, or use arrows

(this key sequence will give you the `TInterval...` screen outside a program)

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

95% unless you give another one. Using the summary stats syntax, the confidence level is also optional - again, the calculator will assume 95%. This means we can rewrite our code above in a simpler manner:

```
:TInterval L1,95  
can be just  
:TInterval
```

```
:TInterval 63,6.2,30,95  
can be  
:TInterval 63,6.2,30
```

Error Conditions

- **ERR:DATA TYPE** occurs if complex numbers are used (in some cases, **ERR:ARGUMENT** is thrown instead).
- **ERR:DIM MISMATCH** occurs if the data and frequency lists aren't the same size.
- **ERR:DOMAIN** occurs in any of the following cases:
 - The confidence level isn't in the range (0 .. 100).
 - The standard deviation isn't positive.
 - The sample size isn't an integer greater than 1.
- **ERR:STAT** occurs if the frequency list's elements aren't integers.

Related Commands

- [2-SampTInt](#)
- [ZInterval](#)
- [2-SampZInt\(](#)

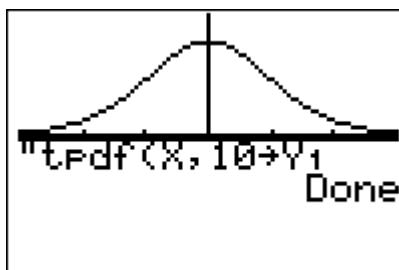
For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/tinterval>

The tpdf(Command

tpdf(is the Student's t probability density function.

Since the t distribution is continuous, the value of tpdf(doesn't represent an actual probability — in fact, one of the only uses for this command is to draw a graph of the bell curve. You could also use it for various calculus purposes, such as finding inflection points.

The command takes two arguments: the first is the value to evaluate it at, and the second is the number of degrees of freedom (so the calculator knows which t distribution to use). As the degrees of freedom get very large, tpdf(approaches [normalpdf\(](#).



Command Summary

Evaluates the Student's t probability density function at a point.

Command Syntax

Formulas

The value of `tpdf`(is given by

$$\text{tpdf}(t, k) = \frac{\Gamma((k+1)/2)}{\sqrt{k\pi}\Gamma(k/2)} (1 + x^2/k)^{-(k+1)/2} \quad (1)$$

Here, Γ is the [Gamma function](#), a generalized version of the factorial.

Related Commands

- [`tcdf`\(](#)
- [`invT`\(](#)
- [`Shade_t`\(](#)

`tpdf(t, df)`

Menu Location

Press:

1. 2ND DISTR to access the distribution menu
2. 4 to select `tpdf`(, or use arrows.

Press 5 instead of 4 on a TI-84+/SE with OS 2.30 or higher.

Calculator Compatibility

TI-83/84+/SE

Token Size

2 bytes

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/tpdf>

The Trace Command

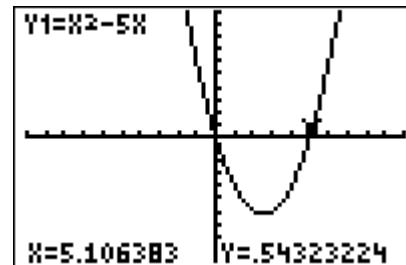
The Trace command displays the graph screen, and allows the user to trace any graphed equations or plots. It works in almost exactly the same way as pressing TRACE does outside a program. When the user presses ENTER, control returns to the program.

When tracing, [ExprOn](#) and [ExprOff](#) affect how the currently-traced equation is displayed, and [CoordOn](#) and [CoordOff](#) affect whether the coordinates of the cursor are displayed ([RectGC](#) and [PolarGC](#) determine the type of coordinates).

Since the ENTER key is already used for exiting, the Trace command lacks some of the functionality of pressing TRACE outside a program, where you can use ENTER to center the graphing window on the cursor.

Advanced Uses

As a side effect, the coordinates of the last point traced are stored to X and Y (as well as R and θ , if you're in [PolarGC](#) mode, and T, θ and n depending on the graphing mode). Also, the window bounds may change if the user traces an equation past the edge of the screen.



Command Summary

Displays the graph screen and allows the user to trace the currently-graphed equations and plots.

Command Syntax

`Trace`

Menu Location

While editing a program, press the TRACE key.

Calculator Compatibility

TI-83/84+/SE

Error Conditions

- **ERR:INVALID** is thrown if this command is used outside a program.

Related Commands

- [Input](#)
- [Select\(](#)
- [DispGraph](#)
- [DispTable](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/trace>

The T-Test Command

T-Test performs a t significance test of a null hypothesis you supply. This test is valid for simple random samples from a population with an unknown standard deviation. In addition, either the population must be normally distributed, or the sample size has to be sufficiently large.

The logic behind a T-Test is as follows: we want to test the hypothesis that the true mean of a population is a certain value (μ_0). To do this, we assume that this "null hypothesis" is true, and calculate the probability that the variation from this mean occurred, under this assumption. If this probability is sufficiently low (usually, 5% is the cutoff point), we conclude that since it's so unlikely that the data could have occurred under the null hypothesis, the null hypothesis must be false, and therefore the true mean μ is not equal to μ_0 . If, on the other hand, the probability is not too low, we conclude that the data may well have occurred under the null hypothesis, and therefore there's no reason to reject it.

In addition to the null hypothesis, we must have an alternative hypothesis as well - usually this is simply that the true mean is **not** μ_0 . However, in certain cases when we have reason to suspect the true mean is less than or greater than μ_0 , we might use a "one-sided" alternative hypothesis, which will state that the true mean $\mu < \mu_0$ or that $\mu > \mu_0$.

As for the T-Test command itself, there are two ways of calling it: you may give it a list of all the sample data, or the necessary statistics about the list - its size, the mean, and the standard deviation.

Token Size

1 byte

```
T-Test
μ≠0
t=14.90711985
P=.139158E-12
x=.5
Sx=.15
n=20
```

Command Summary

Performs a Student's t significance test.

Command Syntax

T-Test μ_0 , //frequency//,
//alternative//, //draw?//
(data list input)

T-Test μ_0 , *sample mean*, *sample std. dev.*, *sample size*, //draw?//
(summary stats input)

Menu Location

While editing a program, press:

1. STAT to access the statistics menu
2. LEFT to access the TESTS submenu
3. 2 to select T-Test, or use arrows

(outside the program editor. this will

In either case, you can indicate what the alternate hypothesis is, by a value of 0, -1, or 1 for the *alternative* argument. 0 indicates a two-sided hypothesis of $\mu \neq \mu_0$, -1 indicates $\mu < \mu_0$, and 1 indicates $\mu > \mu_0$. (in fact, any negative argument will be treated as -1, and any positive argument as 1)

Although you can access the T-Test command on the home screen, via the catalog, there's no need: the T-Test... interactive solver, found in the statistics menu, is much more intuitive to use - you don't have to memorize the syntax.

In either case, it's important to understand the output of T-Test. Here are the meanings of each line:

- The first line, involving μ , is the alternative hypothesis.
- t is the test statistic, the standardized difference between the sample mean and μ_0 . If the null hypothesis is true, it should be close to 0.
- p is the probability that the difference between the sample mean and μ_0 would occur if the null hypothesis is true. When the value is sufficiently small, we reject the null hypothesis and conclude that the alternative hypothesis is true. You should have a cutoff value ready, such as 5% or 1%. If p is lower, you "reject the null hypothesis on a 5% (or 1%) level" in technical terms.
- $x\bar{}$ is the sample mean.
- Sx is the sample standard deviation.
- n is the sample size (not included, but also important, is df , the degrees of freedom, defined as $n-1$)

Sample Problem

According to M&M's advertising, each standard-size bag of M&M's contains an average of 10 blue M&M's. You think that this estimate is low, and that the true average is higher. You decide to test this hypothesis by buying thirty bags of M&M's. You count the number of blue M&M's in each, and store this number to L1.

The value of μ_0 is 10, because you want to test the null hypothesis that there are on average 10 blue M&M's per bag. We want to test the values in L1. Because we want to test if there's actually more than 10 blue M&M's per bag, we have a one-sided alternate hypothesis: $\mu > \mu_0$, which corresponds to an argument of 1. To solve the problem, you'd use this code:

```
:T-Test 10,L1,1
```

Alternatively, you could calculate the mean, standard deviation, and size of your sample, and put those into the command instead. The sample size is 30; let's suppose that the mean was 11.2 and the standard deviation 1.3. The code you'd use is:

```
:T-Test 10,11.2,1.3,30,1
```

You will see the following output:

```
T-Test
```

(select the T-Test... interactive solver)

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

```
 $\mu > 10$ 
z = 5.055900531
p = 1.0857768e-5
x = 11.2
Sx = 1.3
n = 30
```

The most important part of this output is "p=1.0857768e-5". This value of p is much smaller than 1% or 0.01; it's in fact around 0.00001. This is significant on the 1% level, so we reject the null hypothesis and conclude that the alternative hypothesis is true: $\mu > 10$, that is, the average number of blue M&M's in a bag is more than 10.

Advanced Uses

The final argument of T-Test, *draw?*, will display the results in a graphical manner if you put in "1" for it. The calculator will draw the Student's *t* distribution with the correct degrees of freedom, and shade the area of the graph beyond the *t* statistic. In addition, the value of *t* and the value of *p* will be displayed (the value of *p* corresponds to the shaded area). You would make your conclusions in the same way as for the regular output.

As with most other statistical commands, you may use a frequency list in your input (when using the data list syntax).

Optimization

Most of the arguments of the T-Test command have default values, and the argument can be omitted if this value is accepted.

- The *draw?* argument can be omitted if you don't want graphical output, although you could put "0" in as well.
- If the *draw?* argument is omitted, you can omit the *alternative* argument to use a two-sided test ($\mu \neq \mu_0$). If you include the *draw?* argument, you have to include this - otherwise there will be confusion as to what the 5th argument means.
- With data list input, you can always omit the frequency list if you won't be using it.
- With data list input, if the *draw?* and *alternative* arguments are omitted, and your data is in L1, you may omit L1 as well. However, if *alternative* or *draw?* is present, you have to include it, or else the syntax may be confused with the syntax for summary stats input.

The code in the sample problem above can't be optimized, because the *alternative* argument is 1:

```
::T-Test 10, L1, 1
```

However, if we were doing a two-sided test, we could omit the *alternative* and the *list* arguments (since we're testing L1):

```
:T-Test 10, L1, 0  
can be  
:T-Test 10
```

Related Commands

- [2-SampTTest](#)
- [Z-Test](#)
- [2-SampZTest](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/t-test>

The T (Transpose) Command

The T command is used to calculate the transpose of a matrix: it flips a matrix along its main diagonal. This means that the (i,j) th element becomes the (j,i) th element, and vice versa. As a result, the transpose of an M by N matrix is an N by M matrix.

```
[[1,2,3][4,5,6]]
[[1 2 3]
 [4 5 6]]
AnsT
[[1 4]
 [2 5]
 [3 6]]
```

Advanced Uses

In addition to its many uses in linear algebra, the T operation is useful to programmers: with operations such as [Matr▶list\(](#) and [augment\(](#), which normally deal with columns, T allows you to use rows instead. See the "Related Commands" section for the commands that this is useful for.

Related Commands

- [augment\(](#)
- [cumSum\(](#)
- [Matr▶list\(](#)
- [rowSwap\(](#) (and other row operations)

```
[[1,2][3,4]
 [[1 2]
 [3 4]]
AnsT
[[1 3]
 [2 4]]]
```

Command Summary

This command calculates the transpose of a matrix.

Command Syntax

matrix^T

Menu Location

Press:

1. MATRIX (on the 83) or 2nd MATRIX (83+ or higher) to access the Matrix menu.
2. LEFT to access the MATH submenu
3. 2 to select T , or use arrows

Calculator Compatibility

TI-83/84/+/SE

Token Size

[1 byte](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/transpose>

The tvm_Pmt, tvm_I%,

tvm_PV, tvm_N, and tvm_FV Commands

The tvm_VAR commands use the TVM (Time Value of Money) solver to solve for the variable VAR.

They're usually used in programs, since outside a program it's easier to use the interactive solver (the first option in the finance menu).

All five commands can be used by themselves, with no arguments. In that case, they will return the value of VAR solved from the current values of the other finance variables.

If you give them arguments, the values you give will replace the values of the finance variables. You can supply as many or as few arguments as needed, and the finance variables will be replaced in the order: **N**, **I%**, **PV**, **PMT**, **FV**, **P/Y**, **C/Y** (skipping the one you're solving for).

Error Conditions

- **ERR:ITERATIONS** is thrown if the maximum amount of iterations was exceeded in computing I% (this usually means there is no solution)
- **ERR:NO SIGN CHG** is thrown if calculating I% when FV, (N*PMT), and PV all have the same sign.

Related Commands

- [Pmt_End](#)
- [Pmt_Bgn](#)

Command Summary

Solves for the specified finance variable.

Command Syntax

`tvm_Pmt(N,I%,PV,FV,P/Y,C/Y)`

`tvm_I%(N,PV,PMT,FV,P/Y,C/Y)`

`tvm_PV(N,I%,PMT,FV,P/Y,C/Y)`

`tvm_N(I%,PV,PMT,FV,P/Y,C/Y)`

`tvm_FV(N,I%,PV,PMT,P/Y,C/Y)`

All arguments are optional.

Menu Location

On the TI-83, press:

1. 2nd FINANCE to access the finance menu.
2. 2 through 6 to select tvm_Pmt through tvm_FV respectively.

On the TI-83+ and higher, press:

1. APPS to access the applications menu.
2. 1 or ENTER to select Finance...
3. 2 through 6 to select tvm_Pmt through tvm_FV respectively.

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

The UnArchive Command

The UnArchive command moves a variable from the archive (also known as ROM) to RAM. A quick synopsis of the difference between the two:

- Data in the archive cannot be accessed, but it's protected from RAM clears (which may occur during battery removal if not done carefully); also, the archive can hold much more data.
- Data in RAM can be accessed for calculations, but it can also be deleted during a RAM clear or accidentally overwritten by another program.

It is, in general, not recommended to place real variables in the archive (since so many programs use them); also, some variables cannot be archived (see the Archive command for details). Although programs can be archived and unarchived, the Archive and UnArchive commands will not archive or unarchive programs from within a program. For the most part, lists are the only type of variable it makes sense to archive and unarchive in a program.

The UnArchive command doesn't do anything if the variable in question is already in RAM. However, there is no way to test if a variable is in RAM or archive, short of trying to access it and potentially getting an error.

Advanced Uses

The Archive and UnArchive commands can be used in conjunction for saving data as a program exits.

Optimization

The SetUpEditor command is often used in place of the UnArchive command when dealing with lists, for several reasons:

- using SetUpEditor will not prevent the program from working on a TI-83, like UnArchive will
- SetUpEditor will create a list with length 0 if it doesn't exist; UnArchive will throw an error
- SetUpEditor saves space in the program, since it can unarchive more than one list at a time, and doesn't require the little L in front

Error Conditions

- ERR:MEMORY is thrown if there isn't enough memory available in RAM for the variable.
- ERR:VARIABLE is thrown when unarchiving a system variable.



Command Summary

Moves a variable from the archive to RAM.

Command Syntax

`UnArchive variable`

Menu Location

Press:

1. 2nd MEM to access the memory menu
2. 6 to select UnArchive, or use arrows.

Calculator Compatibility

TI-83+/84+/SE

(not available on the regular TI-83)

Token Size

2 bytes

Related Commands

- [Archive](#)
- [DelVar](#)
- [SetUpEditor](#)

See Also

- [Saving Data](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/unarchive>

The uvAxes Command

When uvAxes is enabled, and the calculator is in Seq mode, the equations u and v will be graphed against each other (that is, the points $(u(n),v(n))$ are graphed for the values of n between $nMin$ and $nMax$). With this setting, sequence mode graphs are a bit like parametric mode, except the parameter n is always an integer, and recursive definitions are possible.

The equation w is ignored when in uvAxes mode.

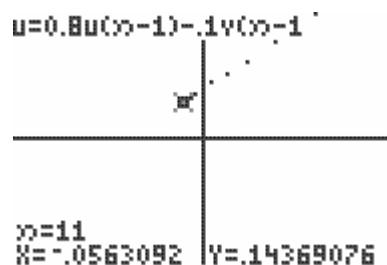
See "Related Commands" for other possibilities of graphing sequences.

Error Conditions

- [ERR:INVALID](#) is thrown if either u or v is undefined.

Related Commands

- [Time](#)
- [Web](#)
- [uwAxes](#)
- [vwAxes](#)



Command Summary

Sets the u and v sequence equations to be graphed against each other.

Command Syntax

uvAxes

Menu Location

When Seq mode is enabled, press:

1. 2nd FORMAT to access the format menu.
2. Use arrows to select uvAxes.

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/uvaxes>

The uwAxes Command

When uwAxes is enabled, and the calculator is in Seq mode, the equations u and w will be graphed against each other (that is, the points $(u(n),w(n))$) are graphed for the values of n between $nMin$ and $nMax$). With this setting, sequence mode graphs are a bit like parametric mode, except the parameter n is always an integer, and recursive definitions are possible.

The equation v is ignored when in uwAxes mode.

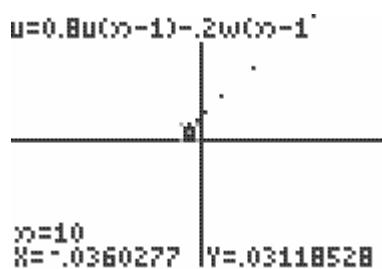
See "Related Commands" for other possibilities of graphing sequences.

Error Conditions

- ERR:INVALID is thrown if either u or w is undefined.

Related Commands

- Time
- Web
- uvAxes
- vwAxes



Command Summary

Sets the u and w sequence equations to be graphed against each other.

Command Syntax

uwAxes

Menu Location

When Seq mode is enabled, press:

1. 2nd FORMAT to access the format menu.
2. Use arrows to select uwAxes.

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/uwaxes>

The variance(Command

The variance(command finds the sample variance of a list, a measure of the spread of a distribution. It takes a list of real numbers as a parameter. For example:

```
:Prompt L1
:Disp "VARIANCE OF L1",variance(L1
```

```
variance({5,4,3,
2,1})  
2.5
```

Command Summary

Advanced Uses

Frequency lists don't need to be whole numbers. Amazing as that may sound, your calculator can handle being told that one element of the list occurs 1/3 of a time, and another occurs 22.7 times. It can even handle a frequency of 0 - it will just ignore that element, as though it weren't there. One caveat, though - if all of the elements occur 0 times, there's no elements actually in the list and your calculator will throw an error.

Formulas

The formula for variance used by this command is:

$$s_n^2 = \frac{1}{N-1} \sum_{i=1}^N (x_i - \bar{x})^2 \quad (1)$$

This is the formula for sample variance. The formula for population variance, which this command does **not** use, varies slightly:

$$\sigma^2 = \frac{1}{N} \sum_{i=1}^N (x_i - \bar{x})^2 \quad (2)$$

If the population variance is required, just multiply the result of variance(with $1 - \frac{1}{N}$.

With frequencies w_i , the formula becomes

$$s_n^2 = \frac{\sum_{i=1}^N w_i(x_i - \bar{x})^2}{\sum_{i=1}^N (w_i) - 1} \quad (3)$$

where \bar{x} is the mean with frequencies included.

Related Commands

- [mean\(](#)
- [median\(](#)
- [stdDev\(](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/variance>

The Vertical Command

Vertical X draws a vertical line from the top of the graph screen to the bottom at X. Vertical is usually

Command Summary

Finds the sample variance of a list.

Command Syntax

`variance(list,[freqlist])`

Menu Location

Press:

1. 2ND LIST to enter the LIST menu.
2. LEFT to enter the MATH submenu.
3. 8 to select variance(, or use arrows.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

(2)

(3)



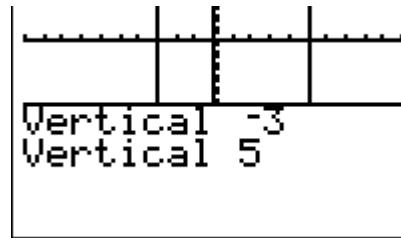
only used to replace a line that stretches the entire length of the graph screen, along with its counterpart [Horizontal](#).

Vertical is affected by the window settings, unlike the [Pxl-](#) commands.

```
:Vertical 5
```

Related Commands

- [Line](#)
- [Horizontal](#)



Command Summary

Draws a vertical line on the graph screen.

Command Syntax

Vertical X

Menu Location

In the program editor:

1. 2nd DRAW to enter the draw menu.
2. 4 to insert the Vertical command, or use arrows and ENTER.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/vertical>

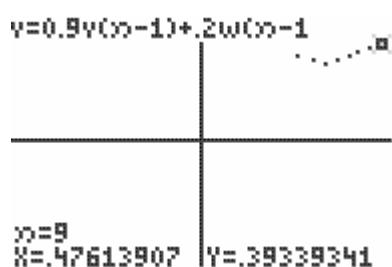
The vwAxes Command

When vwAxes is enabled, and the calculator is in [Seq](#) mode, the equations v and w will be graphed against each other (that is, the points $(v(n),w(n))$ are graphed for the values of n between $nMin$ and $nMax$). With this setting, sequence mode graphs are a bit like [parametric](#) mode, except the parameter n is always an integer, and recursive definitions are possible.

The equation u is ignored when in vwAxes mode.

See "Related Commands" for other possibilities of graphing sequences.

Error Conditions



Command Summary

Sets the v and w sequence equations to be graphed against each other.

- **ERR:INVALID** is thrown if either v or w is undefined.

Related Commands

- Time
- Web
- uvAxes
- uwAxes

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/vwaxes>

The Web Command

In Web mode, sequence equations are graphed as web diagrams. This is a way of visualizing iterations of a function (that is, the sequence $n, f(n), f(f(n)), f(f(f(n))), \dots$ for some function f and starting value n). For this mode to properly work, each sequence equation should be in terms of its previous value only: $u(n)$ should be a function of $u(n-1)$. Referencing other sequence equations, or $u(n-2)$, will yield **ERR:INVALID**; referencing the value n is allowed by the calculator, but makes the result meaningless so you should avoid it.

When you go to the graph screen, the associated function $y=f(x)$ will be graphed. That is, if you define $u(n) = \cos(u(n-1))$, the function $y=\cos(x)$ will be graphed. If you have AxesOn enabled, the line $y=x$ will also be graphed. It's easy to see that the intersection points of the graphs $y=f(x)$ and the line $y=x$ represent the fixed points (points such that $f(x)=x$) of the function.

The web diagram itself will be drawn if you press TRACE or use the Trace command. First you will choose the equation (u, v , or w) to trace; then, by pressing RIGHT repeatedly, the web will be drawn, starting from the initial value $nMin$. In a web diagram, a point $(n, f(n))$ on the graph of $y=f(x)$ is connected by a horizontal segment to the point $(f(n), f(n))$ on the graph of $y=x$, and then by a vertical segment to the point $(f(n), f(f(n)))$ on the graph of

Command Syntax

`vwAxes`

Menu Location

When Seq mode is enabled, press:

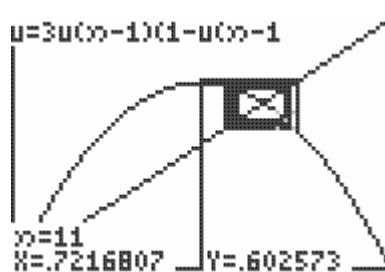
1. 2nd FORMAT to access the format menu.
2. Use arrows to select `vwAxes`.

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes



Command Summary

Sets sequence equations to be graphed as web diagrams.

Command Syntax

`Web`

Menu Location

While in Seq mode, press:

1. 2nd FORMAT to access the format menu.
2. Use arrows and ENTER to select `Web`.

Calculator Compatibility

TI-83/84/+/SE

$y=f(x)$ again; this process is repeated. Each pair of a horizontal and vertical segment represents an added iteration of.

Web diagrams can be used to look at the attracting behavior of fixed points. For example:

1. Graph the equation $u(n)=\cos(u(n-1))$, $u(nMin)=1$ in Web mode, with $Xmin=0$, $Xmax=1$, $Ymin=0$, $Ymax=1$ in the WINDOW menu. You'll see that it has a single fixed point. If you TRACE the graph, the line segments will spiral around into the fixed point, so appears to be attractive.
2. Graph the equation $u(n)=\pi/2\cos(u(n-1))$, $u(nMin)=1$ in Web mode, with $Xmin=0$, $Xmax=\pi/2$, $Ymin=0$, $Ymax=\pi/2$ in the WINDOW menu. This equation looks a lot like the previous one, and also has a single fixed point. However, if you TRACE the graph, the line segments (which start out quite close to the fixed point) will spiral away from it. This intuitively shows that the fixed point of $f(x)=\pi/2\cos(x)$ is not attractive.

See "Related Commands" for other possibilities of graphing sequences.

Error Conditions

- **ERR:INVALID** is thrown if an equation being graphed references other sequence equations or the $n-2$ term.

Related Commands

- Time
- uvAxes
- uwAxes
- vwAxes

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/web>

The While Command

A While loop executes a block of commands between the While and End commands as long as the specified condition is true. The condition is tested at the beginning of the loop (when the End command is encountered), so if the condition is initially false, the block of commands will never get executed. This distinguishes it from the Repeat command.

After each time the While loop is executed, the condition is checked to see if it is still true. If it is, the block of commands is executed again, otherwise the program resumes after the End statement.

Advanced Uses

Token Size

2 bytes

PROGRAM: EXAMPLE
:While 1
:Disp "TI-BASIC
IS KING
:End

Command Summary

Loops through a block of code while the condition is true.

Command Syntax

While *condition*

When using While loops, you have to provide the code to break out of the loop (it isn't built into the loop). If there is no code that ends the loop, then you will have an infinite loop. An infinite loop just keeps executing, until you have to manually exit the loop (by pressing the ON key). In the case that you actually want an infinite loop, you can just use 1 as the condition. Because 1 is always true (based on Boolean logic), the way the calculator sees it, the condition will always be true, and the loop will never end.

```
:While 1  
:statement(s)  
:End
```

Each time the program enters an While block, the calculator uses $35 + (\text{size of the condition})$ bytes of memory to keep track of this. This memory is given back to you as soon as the program reaches End.

This isn't really a problem unless you're low on RAM, or have a lot of nested While statements. However, if you use [Goto](#) to jump out of a While block, you lose those bytes for as long as the program is running — and if you keep doing this, you might easily run out of memory, resulting in [ERR:MEMORY](#).

Optimization

Because the While and Repeat commands are so similar, either one can be used in the same situation, but using one usually results in simpler code than the other. To decide which to use, answer some simple questions about the purpose of the code.

1. Should the code inside the loop be executed at least once? (Alternatively, does the condition use some variable that we first use inside the loop?) If it should, use a Repeat loop. Otherwise, use a While loop.
2. (Only if the previous question doesn't help) Think of the condition based on which the loop keeps going. Is this condition best phrased as "run the loop as long as this is true?" If so, use a While loop. Or is it more like "run the loop until this is true?" If so, Repeat is best.

Example: we want the user to pick a number, but it has to be positive, so we'll keep asking until it is.

1. Yes, we should run the loop once. Otherwise, where will we get the number from? So, we should use the Repeat loop.

```
:Repeat N>0  
:Prompt N  
:End
```

Another example: we want to wait for the user to [press a key](#).

1. We're not going to have any code in the loop, all that the loop will have is a condition. So the answer to question 1 is irrelevant.
2. We can phrase the problem as "run the loop until a key is pressed" or as "run the loop while no key is pressed." However, we have a good way of testing for the former (`getKey`),

statement(s)

End

Menu Location

While editing a program press:

1. PRGM to enter the PRGM menu
2. 5 to choose While, or use arrows
3. 7 to choose End, or use arrows

Calculator Compatibility

TI-83/84+/SE

Token Size

1 byte

while the latter can only be checked with `not(getKey)`. Therefore, it's better to use a Repeat command:

```
:Repeat getKey  
:End
```

Command Timings

While and Repeat loops are identical regarding speed, so that shouldn't be a factor in deciding between them. However, For(loops are much faster at what they do, that is, at going through consecutive values for one variable. You should consider if a For(loop is more appropriate to your situation. If not, choose between a Repeat loop and a While loop.

Error Conditions

- **ERR:INVALID** occurs if this statement is used outside a program.

Related Commands

- [For\(](#)
- [Repeat](#)
- [If](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/while>

The xor Command

The third and final binary operator is a little trickier, but it has the coolest name. **xor** takes two numbers of expressions and checks to see if *exactly one* is True. If both are True or both are False, it returns 0.

```
1 xor 0  
      1  
  
:2 xor (3 xor 0)      (after evaluatin  
      0  
  
:0 xor (1-1)^2  
      0
```

$2+2=4$	1
$2+2=5$	0
$2+2=4 \text{ xor } 2+2=5$	1

Command Summary

Returns the truth value of *value1* or *value2*, but not both, being true.

Command Syntax

value1 xor value2

Menu Location

Press:

1. 2nd TEST to access the test menu.

Related Commands

- [and](#)
- [or](#)
- [not\(](#)

2. RIGHT to access the LOGIC submenu.
3. 3 to select xor, or use arrows and ENTER.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/xor>

The ZBox Command

The ZBox command allows the user to select a smaller window within the current graphing window to zoom in to. To select the window, use the arrow keys and ENTER to select one corner of the window; then as you use the arrow keys and ENTER to select the other corner, a rectangle of the window's dimensions will be shown.

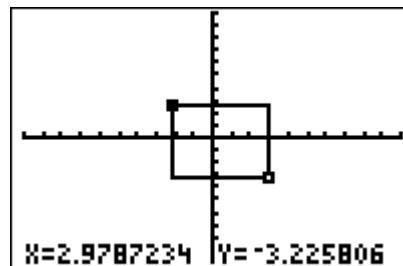
It's not recommended to use this in most programs, because entering an empty window (with no width or no height) will cause an error and exit the program without letting it clean up.

Error Conditions

- **ERR:INVALID** occurs if this command is used outside a program.
- **ERR:ZOOM** is thrown if an empty window is selected (with no width or no height)

Related Commands

- [Zoom In](#)
- [Zoom Out](#)



Command Summary

Zooms in to a smaller graphing window defined by a box drawn by the user.

Command Syntax

ZBox

Menu Location

Press:

1. ZOOM to access the zoom menu.
2. ENTER to select ZBox.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

The ZDecimal Command

The ZDecimal command makes the following changes to the window variables:

- $X_{\min} = -4.7$
- $X_{\max} = 4.7$
- $X_{\text{scl}} = 1$
- $Y_{\min} = -3.1$
- $Y_{\max} = 3.1$
- $Y_{\text{scl}} = 1$

Because of the dimensions of the graph screen (95 by 63, when you remember that the last row and column aren't used), this has the useful effect that every pixel has round X- and Y-coordinates with at most one decimal digit. Also, the screen has correct proportions: a specific distance in the X direction is the same number of pixels in length as the same distance in the Y direction. This makes the window dimensions created by ZDecimal a friendly window (the ZInteger and ZSquare commands also have this effect, but in slightly different ways)

Advanced Uses

Using the ZDecimal command prevents gaps in certain graphs, and makes sure vertical asymptotes with integer coordinates are graphed correctly. Also, circles will be drawn as actual circles with this graphing window(unlike other windows, with which they might appear stretched).

The values given for X_{\min} , X_{\max} , etc. above are only correct for the Full mode setting (which is the default, and the most common setting). In Horiz and G-T modes, the values will be different, preserving the property that two pixels next to each other differ in coordinates by 0.1:

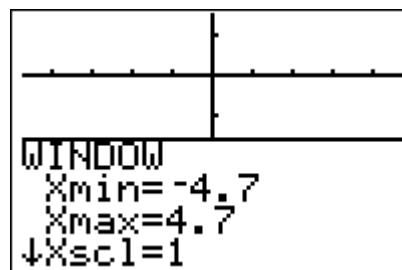
- $Y_{\min} = -1.5$ and $Y_{\max} = 1.5$ in Horiz mode (X_{\min} and X_{\max} are the same)
- $Y_{\min} = -2.5$ and $Y_{\max} = 2.5$ in G-T mode, while $X_{\min} = -2.3$ and $X_{\max} = 2.3$

Error Conditions

- ERR:INVALID occurs if this command is used outside a program.

Related Commands

- ZInteger
- ZSquare



Command Summary

Zooms to a friendly window where all pixels have simple coordinates.

Command Syntax

ZDecimal

Menu Location

Press:

1. ZOOM to access the zoom menu.
2. 4 to select ZDecimal, or use arrows and ENTER.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

- [ZStandard](#)

See Also

- [Friendly Graphing Windows](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/zdecimal>

The ZInteger Command

When ZInteger is chosen as a menu option outside a program, it asks for a point on the graph screen. This point's coordinates are rounded to the nearest integer point. Then the [window variables](#) are changed so the window is centered at this point, and so that the coordinates of every pixel are integers. ΔX and ΔY , the distances between two pixels next to each other, are both 1.

The above process modifies X_{\min} , X_{\max} , Y_{\min} , and Y_{\max} . $X_{\text{sc}}l$ and $Y_{\text{sc}}l$ are also modified: both are set to 10. No other variables are modified (unless you count ΔX and ΔY , which are affected as they are defined).

The ZInteger command (usable in a program only) has a slightly different effect: instead of allowing you to choose a point, it automatically uses the point that is the current center.

Advanced Uses

A graph window commonly used in programming can be created by using the [ZStandard](#) and [ZInteger](#) commands:

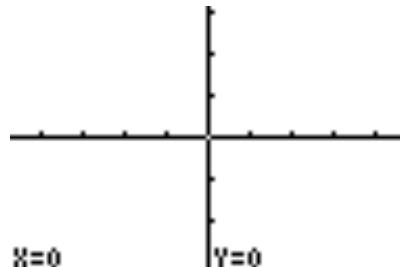
```
:ZStandard
:ZInteger
```

Error Conditions

- [ERR:INVALID](#) occurs if this command is used outside a program.

Related Commands

- [ZDecimal](#)
- [ZStandard](#)
- [ZSquare](#)



Command Summary

Zooms to a square window with all-integer coordinates.

Command Syntax

ZInteger

Menu Location

Press:

1. ZOOM to access the zoom menu.
2. 8 to select ZInteger, or use arrows and ENTER.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

See Also

- [Friendly Graphing Windows](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/zinteger>

The ZInterval Command

The ZInterval command calculates a confidence interval for the mean value of a population, at a specific confidence level: for example, if the confidence level is 95%, you are 95% certain that the mean lies within the interval you get. Use ZInterval when you have a single variable to analyze, and you already know the standard deviation. The ZInterval assumes that your distribution is normal, but it will work for other distributions if the sample size is large enough.

There are two ways to call the ZInterval command: by supplying it with needed sample statistics (mean and sample size), or by entering a list and letting the calculator work the statistics out. In either case, you will need to enter the standard deviation and desired confidence level as well.

Sample Problem

You want to know the average height of a student at your school. You haven't asked everyone, but you took a random sample of 30 people and found out their height (and stored it to L₁). You've read in your textbook that the standard deviation of teenagers' heights is usually 6 inches. You've decided to use a 95% confidence interval.

Since the syntax for entering a data list is ZInterval *std. deviation, list, confidence level*, the code would look like:

```
:ZInterval 6,L1,.95  
you can also use  
:ZInterval 6,L1,.95
```

Alternatively, you could calculate the mean and sample size and enter those instead. The sample size in this case is 30; let's say the mean was 63 inches. The syntax for entering statistics is ZInterval *std. deviation, mean, sample size, confidence level*, so your code would look like:

```
ZInterval  
(-.8772,.47718)  
x=.2  
n=50
```

Command Summary

Using either already-calculated statistics, or a data set, computes a Z confidence interval.

Command Syntax

ZInterval *std. deviation,*
[//frequency//] *level//*
(data list input)

ZInterval *std. deviation, mean,*
sample size, level//
(summary stats input)

Menu Location

When editing a program, press:

1. STAT to access the statistics menu
2. LEFT to access the TESTS submenu
3. 7 to select ZInterval, or use arrows

(this key sequence will give you the ZInterval... screen outside a program)

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

```
:ZInterval 6,63,30,.95  
you can also use  
:ZInterval 6,63,30,.95
```

Of course, the main use of the ZInterval command is in a program. While you can enter the ZInterval command on the home screen as well (just look in the catalog for it), it would probably be easier to select ZInterval... from the STAT>TEST menu (see the sidebar).

Advanced Uses

As with most other statistical commands, you can enter a second list after the data list, to add frequencies (only with the data list syntax, of course). The frequency list must contain non-negative real numbers, and can't be all 0.

Optimization

Using the data list syntax, all items but the standard deviation are optional: the calculator will assume you want to use L1 for your data unless another list is supplied, and that the confidence level you want is 95% unless you give another one. Using the summary stats syntax, the confidence level is also optional - again, the calculator will assume 95%. This means we can rewrite our code above in a simpler manner:

```
:ZInterval 6,L1,95  
can be  
:ZInterval 6
```

```
:ZInterval 6,63,30,95  
can be  
:ZInterval 6,63,30
```

Related Commands

- [2-SampZInt\(](#)
- [TInterval](#)
- [2-SampTInt](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/zinterval>

The ZoomFit Command

The ZoomFit zooms to the smallest window that contains all points of the currently graphed equations. In Func mode, this means that it calculates the minimum and maximum Y-value for the current Xmin to Xmax range, and sets Ymin and Ymax to those values (Xmin and Xmax remain unchanged). In other graphing modes, this process is done for both X and Y over the range of T, θ , or n .

ZOOM MEMORY
1:ZBox
2:Zoom In
3:Zoom Out
4:ZDecimal
5:ZSquare
6:ZStandard
7:ZTrig

Optimization

When graphing an equation with ZoomFit, the calculator will first calculate all points to find the minimum and maximum, then calculate all the points again to graph it. This can be time consuming if the equation is very complicated, and in that case doing part of the process manually might be faster if you reuse the points.

Error Conditions

- **ERR:INVALID** is thrown if this command is used outside a program (although the menu option, of course, is fine).
- **ERR:WINDOW RANGE** is thrown when the window ends up being empty (if the function is constant, for example)

Related Commands

- [ZoomStat](#)
- [ZBox](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/zoomfit>

The Zoom In Command

Outside a program, the Zoom In tool allows you to pick a point on the graph screen and change the graphing window to a smaller one centered at that point. The Zoom In command, used in a program, also changes the graphing window to a smaller one, but doesn't let you pick a point — it uses the center of the screen.

The variables XFact and YFact are used to determine how much the graphing window changes: the total width of the screen, $X_{max} - X_{min}$, is divided by XFact, and the total height, $Y_{max} - Y_{min}$, is divided by YFact. Because you can't store a value less than 1 to either of these variables, the screen is guaranteed to get no larger.

Aside from X_{min} , X_{max} , Y_{min} , and Y_{max} , no window variables are modified by this command (although ΔX and ΔY change as they are defined).

Error Conditions

Command Summary

Zooms to a graphing window that works for the currently graphed equations.

Command Syntax

ZoomFit

Menu Location

Press:

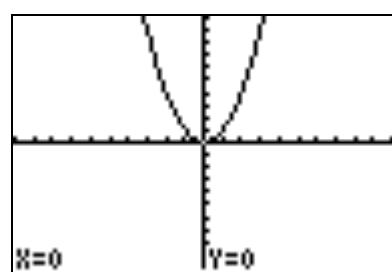
1. ZOOM to access the zoom menu.
2. 0 to select ZoomFit, or use arrows and ENTER.

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes



Command Summary

Zooms to a smaller graphing window with the same centre.

Command Syntax

Zoom In

Menu Location

Press:

- **ERR:INVALID** occurs if this command is used outside a program.
- **ERR:WINDOW RANGE** is thrown if the window is zoomed in beyond the level of the calculator's precision.

Related Commands

- Zoom Out
- ZBox

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/zoom-in>

The Zoom Out Command

Outside a program, the Zoom Out tool allows you to pick a point on the graph screen and change the graphing window to a larger one centered at that point. The Zoom Out command, used in a program, also changes the graphing window to a larger one, but doesn't let you pick a point — it uses the center of the screen.

The variables XFact and YFact are used to determine how much the graphing window changes: the total width of the screen, $X_{max} - X_{min}$, is multiplied by XFact, and the total height, $Y_{max} - Y_{min}$, is multiplied by YFact. Because you can't store a value less than 1 to either of these variables, the screen is guaranteed to get no smaller.

Aside from X_{min} , X_{max} , Y_{min} , and Y_{max} , no window variables are modified by this command (although ΔX and ΔY change as they are defined).

Error Conditions

- **ERR:INVALID** occurs if this command is used outside a program.
- **ERR:ZOOM** is thrown if an overflow occurs calculating the new window dimensions (the window is too big)

Related Commands

- Zoom In
- ZBox

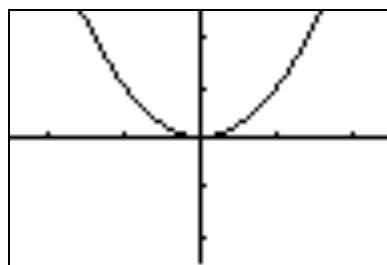
1. ZOOM to access the zoom menu.
2. 2 to select Zoom In, or use arrows and ENTER.

Calculator Compatibility

TI-83/84/+SE

Token Size

1 byte



Command Summary

Zooms to a larger graphing window with the same center.

Command Syntax

Zoom Out

Menu Location

Press:

1. ZOOM to access the zoom menu.
2. 3 to select Zoom Out, or use arrows and ENTER.

Calculator Compatibility

TI-83/84/+SE

Token Size

1 byte

The ZoomRcl Command

The ZoomRcl command restores a backup of the window settings previously saved by ZoomSto — this backup is stored in special variables found in the VARS>Zoom... menu, which are distinguished by a Z in front of their name. For example, Xmin is restored from ZXmin, PlotStart is restored from ZPlotStart, etc.

Only those settings are restored that apply to the current graphing mode (that is, those that you can see in the window screen). And if no backup had been made, then the default settings are restored to (see ZStandard).

One source of confusion with this command can be the fact that ZoomSto and ZoomRcl only deal with the current graphing mode (and don't touch settings from other graphing modes), but some window variables are shared by graphing modes. So some saved zoom variables only applicable to one mode, such as ZTmin, can be from older saves than those applicable to all modes, such as ZXmin.

Error Conditions

- **ERR:INVALID** occurs if this command is used outside a program (but not if the menu option is used, of course).

Related Commands

- [ZoomSto](#)
- [ZPrevious](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/zoomrcl>

The ZoomStat Command

The ZoomStat command zooms to a graphing window that accurately represents all the currently defined stat plots (see the PlotN(commands). You can think of it as ZoomFit, but for plots rather than equations.

The specific function of the command is as follows: first, the minimum and maximum X and Y

Command Summary

Recalls window settings previously saved with ZoomSto.

Command Syntax

ZoomRcl

Menu Location

Press:

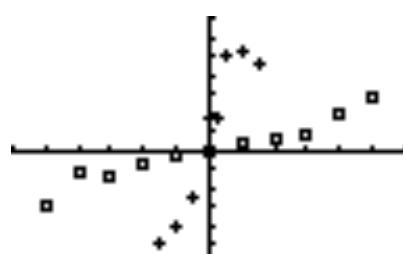
1. ZOOM to access the zoom menu.
2. RIGHT to access the MEMORY submenu.
3. 3 to select ZoomRcl, or use arrows and ENTER.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte



coordinates that stat plots will be using are calculated. X_{\min} , X_{\max} , Y_{\min} , and Y_{\max} are calculated to fit all these coordinates plus a padding on either side. The padding is 10% of the unpadded range on the left and right (for X_{\min} and X_{\max}), and 17% of the unpadded range on the top and bottom (for Y_{\min} and Y_{\max}).

Of course, the exact function varies slightly with the type of plot defined. For example, Y_{\min} and Y_{\max} will not be affected by Boxplot and Modboxplot plots, since they ignore Y-coordinates when graphing. Also, Histogram fitting is a bit trickier than others. X_{sc1} and Y_{sc1} also are changed for histograms, though not for the other plots.

For all plots except Histogram, ZoomStat will create a window with $X_{\min}=X_{\max}$ (or $Y_{\min}=Y_{\max}$) if the X range (or Y range) of the data is 0. This will throw an ERR:WINDOW RANGE. If a Histogram causes this error, though, ERR:STAT is thrown, and then when you access the graphscreen ERR:WINDOW RANGE will occur.

Error Conditions

- ERR:INVALID is thrown if this command is using outside a program (although the menu option, of course, is fine).
- ERR:STAT is thrown when trying to ZoomFit to a Histogram with only one distinct number in the data list.
- ERR:WINDOW RANGE is thrown when the window ends up being empty.

Related Commands

- ZoomFit
- ZBox

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/zoomstat>

The ZoomSto Command

The ZoomSto command backs up all window settings applicable to the current graphing mode (those that are shown in the WINDOW menu) to backup variables used specifically for this command. These backup variables are found in the VARS>Zoom... menu, and are distinguished by a Z in front of their name. For example, X_{\min} is backed up to ZX_{\min} , PlotStart is backed up to $ZPlotStart$, etc.

Using ZoomRcl, these backup variables can be

Command Summary

Zooms to a graphing window which works for all currently selected plots.

Command Syntax

ZoomStat

Menu Location

Press:

1. ZOOM to access the zoom menu.
2. 9 to select ZoomStat, or use arrows and ENTER.

Calculator Compatibility

TI-83/84/+SE

Token Size

1 byte

Command Summary

Saves the current window settings.

Command Syntax

ZoomSto

Menu Location

used to overwrite the current window settings, recalling the saved window.

One source of confusion with this command can be the fact that ZoomSto and ZoomRcl only deal with the current graphing mode (and don't touch settings from other graphing modes), but some window variables are shared by graphing modes. So some saved zoom variables only applicable to one mode, such as ZTmin, can be from older saves than those applicable to all modes, such as ZXmin.

Error Conditions

- **ERR:INVALID** occurs if this command is used outside a program (but not if the menu option is used, of course).

Related Commands

- [ZoomRcl](#)
- [ZPrevious](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/zoomsto>

The ZPrevious Command

The ZPrevious command (and menu option) restore the window variables Xmin, Xmax, Xscl, Ymin, Ymax, and Yscl to the values they had before the last zoom command. This means, of course, that using ZPrevious a second time will cancel its effects.

Since no variables that are specific to the current graphing mode are changed, ZPrevious doesn't always achieve the effect of reversing the previous zoom command. For example, in Polar graphing mode, ZStandard will set θ_{min} and θ_{max} to 0 and 2π respectively. However, even if they were different before ZStandard, ZPrevious will not restore these settings. Also, ZPrevious doesn't notice if you change the window settings directly (by storing to the window variables).

Unlike ZoomSto and ZoomRcl, the values that ZPrevious uses aren't made available in any sort of variable.

Optimization

Using StoreGDB and RecallGDB is an excellent way to back up graph settings so a program doesn't

Press:

1. ZOOM to access the zoom menu.
2. RIGHT to access the MEMORY submenu.
3. 2 to select ZoomSto, or use arrows and ENTER.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

```
WINDOW
Xmin=-10
Xmax=10
Xscl=1
Ymin=-10
Ymax=10
Yscl=1
Xres=1
```

Command Summary

Restores the basic window settings as they were before the last zoom command.

Command Syntax

ZPrevious

Menu Location

Press:

1. ZOOM to access the zoom menu.
2. RIGHT to access the

modify them. However, if all you're doing is changing the window variables with one Zoom command, you can simply use ZPrevious at the end of the program instead.

Error Conditions

- **ERR:INVALID** occurs if this command is used outside a program (but not if the menu option is used, of course).

Related Commands

- [ZoomSto](#)
- [ZoomRcl](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/zprevious>

The ZSquare Command

The ZSquare command changes the window variables X_{\min} and X_{\max} , or Y_{\min} and Y_{\max} , so that $\Delta X = \Delta Y$, preserving all other settings and the coordinate of the center of the screen. This ensures that a numerical distance on the graphscreen has the same physical length on the calculator display, no matter if it's vertical, horizontal, or diagonal. Probably the most obvious effect of this change is that circles (whether graphed with an equation or drawn with the [Circle\(](#) command) are actually circles and not ovals.

When determining which of X_{\min} and X_{\max} or Y_{\min} and Y_{\max} to change, the command picks the ones that would be increased, and not decreased. This way, the window can never get smaller.

Note that [ZDecimal](#), [ZInteger](#), and to an extent [ZTrig](#) already have the same proportions, and don't require a ZSquare command to follow them.

Advanced Uses

ZSquare can be useful in setting up a friendly window.

Error Conditions

- **ERR:INVALID** occurs if this command is used outside a program.

MEMORY submenu.

3. ENTER or 1 to select ZPrevious.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

Circle(0,0,5)■

Command Summary

Zooms to a square window.

Command Syntax

ZSquare

Menu Location

Press:

1. ZOOM to access the zoom menu.
2. 5 to select ZSquare, or use arrows and ENTER.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

Related Commands

- [ZDecimal](#)
- [ZInteger](#)
- [ZStandard](#)

See Also

- [Friendly Graphing Windows](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/zsquare>

The ZStandard Command

The ZStandard command resets all window variables found in the Window screen to their default values. This means that, unlike the other zoom commands, ZStandard can affect variables other than Xmin, Xmax, Ymin, and Ymax. However, it will only affect variables that have a purpose in the current graphing mode. Here are the default values set by ZStandard:

In all modes:

- Xmin=-10
- Xmax=10
- Xscl=1
- Ymin=-10
- Ymax=10
- Yscl=1

Only in Func mode:

- Xres=1

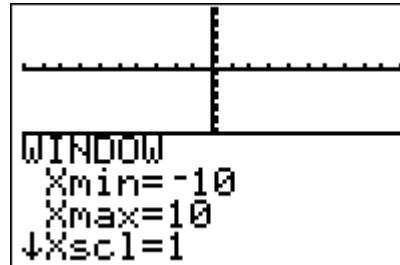
Only in Param mode:

- Tmin=0
- Tmax= 2π (in Radian mode) or 360 (in Degree mode)
- Tstep= $\pi/24$ (in Radian mode) or 7.5 (in Degree mode)

Only in Polar mode:

- $\theta_{\text{min}}=0$
- $\theta_{\text{max}}=2\pi$ (in Radian mode) or 360 (in Degree mode)
- $\theta_{\text{step}}=\pi/24$ (in Radian mode) or 7.5 (in Degree mode)

Only in Seq mode:



Command Summary

Restores window settings to the default.

Command Syntax

ZStandard

Menu Location

Press:

1. ZOOM to access the zoom menu.
2. 6 to select ZStandard, or use arrows and ENTER.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

- $nMin=1$
- $nMax=10$
- $\text{PlotStart}=1$
- $\text{PlotStep}=1$

These settings are often useful as a "lowest common denominator" that will work fairly well for all graphs.

Advanced Uses

`ZStandard` is often used before commands such as `ZSquare` or `ZInteger` in programs. This serves two purposes: it makes sure that the center of the screen for `ZSquare` and `ZInteger` is $(0,0)$, and it ensures that the graph screen is cleared without having to resort to `ClrDraw` (because with two different zooms in a row, the window settings have to change at least once, which means the graph will have to be regraphed)

Error Conditions

- **ERR:INVALID** occurs if this command is used outside a program.

Related Commands

- `ZSquare`
- `ZInteger`

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/zstandard>

The Z-Test(Command

`Z-Test(` performs a z significance test of a null hypothesis you supply. This test is valid for simple random samples from a population with a known standard deviation. In addition, either the population must be normally distributed, or the sample size has to be sufficiently large.

The logic behind a Z-Test is as follows: we want to test the hypothesis that the true mean of a population is a certain value (μ_0). To do this, we assume that this "null hypothesis" is true, and calculate the probability that the variation from this mean occurred, under this assumption. If this probability is sufficiently low (usually, 5% is the cutoff point), we conclude that since it's so unlikely that the data could have occurred under the null hypothesis, the null hypothesis must be false, and therefore the true mean μ is not equal to μ_0 . If, on the other hand, the probability is not too low, we conclude that the data may well have occurred under the null hypothesis, and therefore there's no reason to reject it.

```
Z-Test
μ≠0
z=2.371708245
P=.0177060391
x̄=1.5
n=10
```

Command Summary

Performs a z significance test.

Command Syntax

`Z-Test(μ_0 , σ , //frequency//,
//alternative//, //draw?//
(data list input)`

`Z-Test(μ_0 , σ , sample mean, sample size, //draw?//
(summary stats input)`

In addition to the null hypothesis, we must have an alternative hypothesis as well - usually this is simply that the true mean is **not** μ_0 . However, in certain cases when we have reason to suspect the true mean is less than or greater than μ_0 , we might use a "one-sided" alternative hypothesis, which will state that the true mean $\mu < \mu_0$ or that $\mu > \mu_0$.

As for the Z-Test(command itself, there are two ways of calling it: you may give it a list of all the sample data, or the necessary statistics about the list - its size, and the mean. In either case, you can indicate what the alternate hypothesis is, by a value of 0, -1, or 1 for the *alternative* argument. 0 indicates a two-sided hypothesis of $\mu \neq \mu_0$, -1 indicates $\mu < \mu_0$, and 1 indicates $\mu > \mu_0$.

Although you can access the Z-Test(command on the home screen, via the catalog, there's no need: the Z-Test... interactive solver, found in the statistics menu, is much more intuitive to use - you don't have to memorize the syntax.

In either case, it's important to understand the output of Z-Test. Here are the meanings of each line:

- The first line, involving μ , is the alternative hypothesis.
- z is the test statistic, the standardized difference between the sample mean and μ_0 . If the null hypothesis is true, it should be close to 0.
- p is the probability that the difference between the sample mean and μ_0 would occur if the null hypothesis is true. When the value is sufficiently small, we reject the null hypothesis and conclude that the alternative hypothesis is true. You should have a cutoff value ready, such as 5% or 1%. If p is lower, you "reject the null hypothesis on a 5% (or 1%) level" in technical terms.
- $x\bar{}$ is the sample mean.
- Sx is the sample standard deviation. This isn't actually used in any calculations, and will only be shown for data list input.
- n is the sample size.

Sample Problem

According to M&M's advertising, each standard-size bag of M&M's contains an average of 10 blue M&M's with a standard deviation of 2 M&M's. You think that this estimate is low, and that the true average is higher. You decide to test this hypothesis by buying thirty bags of M&M's. You count the number of blue M&M's in each, and store this number to L1.

The value of μ_0 is 10, because you want to test the null hypothesis that there are on average 10 blue M&M's per bag. The value of σ is 2. We want to test the values in L1. Because we want to test if there's actually more than 10 blue M&M's per bag, we have a one-sided alternate hypothesis: $\mu > \mu_0$, which corresponds to an argument of 1. To solve the problem, you'd use this code:

```
: Z-Test(10, 2, L1, 1)
```

Menu Location

While editing a program, press:

1. STAT to access the statistics menu
2. LEFT to access the TESTS submenu
3. ENTER to select Z-Test(

(outside the program editor, this will select the Z-Test... interactive solver)

Calculator Compatibility

TI-83/84/+SE

Token Size

2 bytes

Alternatively, you could calculate the mean and sample size of your sample, and put those into the command instead. The sample size is 30; let's suppose that the mean was 11.2. The code you'd use is:

```
: Z-Test(10, 2, 11.2, 30, 1
```

You will see the following output:

```
Z-Test  
μ>10  
z=3.286335345  
p=5.0755973e-4  
x=11.2  
n=30
```

The most important part of this output is "p=5.0755973e-4". This value of p is much smaller than 1% or 0.01; it's in fact around 0.0005. This is significant on the 1% level, so we reject the null hypothesis and conclude that the alternative hypothesis is true: $\mu > 10$, that is, the average number of blue M&M's in a bag is more than 10.

Advanced Uses

The final argument of Z-Test(, *draw?*, will display the results in a graphical manner if you put in "1" for it. The calculator will draw the **standard** normal curve, and shade the area of the graph beyond the z statistic. In addition, the value of z and the value of p will be displayed (the value of p corresponds to the shaded area). You would make your conclusions in the same way as for the regular output.

As with most other statistical commands, you may use a frequency list in your input (when using the data list syntax).

Optimization

Most of the arguments of the Z-Test(command have default values, and the argument can be omitted if this value is accepted.

- The *draw?* argument can be omitted if you don't want graphical output, although you could put "0" in as well.
- If the *draw?* argument is omitted, you can omit the *alternative* argument to use a two-sided test ($\mu \neq \mu_0$). If you include the *draw?* argument, you have to include this - otherwise there will be confusion as to what the 5th argument means.
- With data list input, you can always omit the frequency list if you won't be using it.
- With data list input, if the *draw?* and *alternative* arguments are omitted, and your data is in L1, you may omit L1 as well. However, if *alternative* or *draw?* is present, you have to include it, or else the syntax may be confused with the syntax for summary stats input.

The code in the sample problem above can't be optimized, because the *alternative* argument is 1:

```
:: Z-Test(10, 2, L1, 1
```

However, if we were doing a two-sided test, we could omit the *alternative* and the *list* arguments (since we're testing L1):

```
:Z-Test(10,2,L1,0  
can be  
:Z-Test(10,2
```

Related Commands

- [2-SampZTest\(](#)
- [T-Test](#)
- [2-SampTTest](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/z-test>

The ZTrig Command

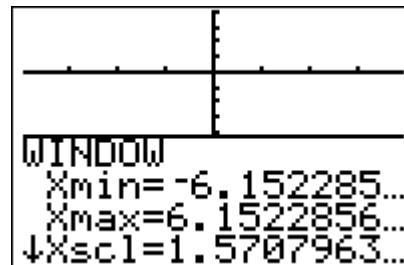
The ZTrig command sets the screen to a special friendly window useful for trigonometric calculations. Unlike the ZDecimal and ZInteger commands, for which the distance between two pixels is a short decimal or integer, ZTrig sets the horizontal distance between two pixels to be $\pi/24$ (in Radian mode) or 7.5 (in Degree mode). The specific changes ZTrig makes are:

- $X_{\min} = -352.5^\circ$ or $-47/24\pi^\circ$
- $X_{\max} = 352.5^\circ$ or $47/24\pi^\circ$
- $X_{\text{scl}} = 90^\circ$ or $\pi/2^\circ$
- $Y_{\min} = -4$
- $Y_{\max} = 4$
- $Y_{\text{scl}} = 1$

Although this window is not quite square (and therefore, distances in the X and Y direction are not exactly equally proportioned), it is quite close, when in Radian mode. In a square window (such as the output of ZSquare), Y_{\max} would have to be $31/24\pi$, which is approximately 4.05789. As you can see, the value of 4 that ZTrig uses is not too far off.

Advanced Uses

In theory, ZTrig should be quite useful in graphing trigonometric functions, since the calculated points would fall exactly on important angles; for example, it would graph the asymptotes of $Y = \tan(X)$ correctly. This is actually only true when in Degree mode. In Radian mode, due to round-off error, the pixels far away from the origin do not *exactly* correspond to rational multiples of π . For example, the pixel which



Command Summary

Zooms to a trigonometry-friendly window.

Command Syntax

ZTrig

Menu Location

Press:

1. ZOOM to access the zoom menu.
2. 7 to select ZTrig, or use arrows and ENTER.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

was supposed to correspond to $\pi/2$ actually has a value of $.5000000001\pi$, which is enough to make this command mostly useless.

Although in G-T mode, the size that the graph takes up on the screen is different, ZTrig uses the same values, unlike ZDecimal.

Error Conditions

- ERR:INVALID occurs if this command is used outside a program (but not if the menu option is used, of course).

Related Commands

- ZDecimal
- ZInteger
- ZStandard

See Also

- Friendly Graphing Windows

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/ztrig>

The Δ List(Command

Δ List(calculates the differences between consecutive terms of a list, and returns them in a new list.

```
 $\Delta$ List({0,1,4,9,16,25,36})  
{1 3 5 7 9 11}
```

```
 $\Delta$ List({1,4,9,16,  
25  
}{3 5 7 9})  
 $\Delta$ List({3,5,7,9  
}{2 2 2})
```

Advanced Uses

The Δ List(command is very nearly the inverse of the cumSum(command, which calculates the cumulative sums of a list. For any list, Δ List(cumSum(*list*)) will return the same list, but without its first element:

```
 $\Delta$ List(cumSum({1,2,3,4,5,6,7}))  
{2 3 4 5 6 7}
```

Removing the first element would otherwise be a difficult procedure involving the seq(command, so this is a useful trick to know.

If a list is sorted in ascending order, $\min(\Delta$ List(*list*))

Command Summary

Calculates the differences between consecutive terms of a list.

Command Syntax

Δ List(*list*)

Menu Location

Press:

1. 2nd LIST to access the list menu.
2. RIGHT to access the OPS submenu.
3. 7 to select Δ List(, or use arrow

will return 0 (false) if there are repeating values in the list, and a value corresponding to true if they are all distinct.

Related Commands

- sum(
- cumSum(
- augment(

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/deltalist>

allows.

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

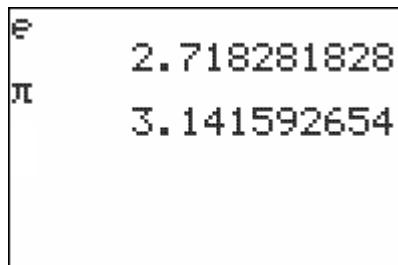
The π Command

The π constant on the calculator approximates the mathematical constant π (pi), defined as the ratio of a circle's circumference to its diameter. It's fairly important in many areas of mathematics, but especially trigonometry. Probably the most common use of π on the TI-83 series calculators is for dealing with angles in radian form.

The approximate value of π stored on the calculator is 3.1415926535898.

Related Commands

- e
- L



A screenshot of a TI-83 calculator's display. The top line shows the value of the mathematical constant e as 2.718281828. The bottom line shows the value of π as 3.141592654.

Command Summary

The mathematical constant π

Command Syntax

π

Menu Location

Press 2nd π to paste the π symbol.

Calculator Compatibility

TI-83/84/+/SE

Token Size

1 byte

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/pi>

The Σ Int(Command

The Σ Int(command calculates, for an amortization schedule, the interest over a range of payments: the



A screenshot of a TI-83 calculator's display showing the command `30*12→N:8/12→I%:` entered into the command line.

portion of those payments that went toward paying interest. Its two required arguments are *payment1* and *payment2*, which define the range of payments we're interested in. However, it also uses the values of the finance variables PV, PMT, and I% in its calculations.

The optional argument, *roundvalue*, is the number of digits to which the calculator will round all internal calculations. Since this rounding affects further steps, this isn't the same as using `round(` to round the result of $\Sigma\text{Int}($ to the same number of digits.

Usually, you will know the values of **N**, PV, and I%, but not PMT. This means you'll have to use the finance solver to solve for PMT before calculating $\Sigma\text{Int}($; virtually always, FV will equal 0.

Sample Problem

Imagine that you have taken out a 30-year fixed-rate mortgage. The loan amount is 0000, and the annual interest rate (APR) is 8%. Payments will be made monthly. How much of the amount that was paid in the first five years went towards interest?

We know the values of **N**, I%, and PV, though we still need to convert them to monthly values (since payments are made monthly). **N** is 30×12 , and I% is $8/12$. PV is just 100000.

Now, we use the finance solver to solve for PMT. Since you intend to pay out the entire loan, FV is 0. Using either the interactive TVM solver, or the `tvm_Pmt` command, we get a value of about -3.76 for PMT.

We are ready to use $\Sigma\text{Int}($. We are interested in the payments made during the first five years; that is, between the 1st payment and the $5 \times 12 = 60$ th payment. $\Sigma\text{Int}(1,60)$ gives us the answer: -095.73 (the negative sign simply indicates the direction of cash flow)

Formulas

$\Sigma\text{Int}($ is calculated in terms of `ΣPrn(`, for which a recurrence exists. Since the total amount paid during an interval is known (it's the payment size, multiplied by the number of payments), we can subtract `ΣPrn(` from this total to get $\Sigma\text{Int}($:

$$\Sigma\text{Int}(n_1, n_2) = (n_2 - n_1 + 1) \text{ PMT} - \Sigma\text{Prn}(n_1, n_2) \quad (1)$$

Error Conditions

- **ERR:DOMAIN** is thrown if either payment number is negative or a decimal.

100000 → PV: 0 → FV	0
tvm_Pmt → PMT	-733.7645739
ΣInt(1,60	-39095.73115

Command Summary

For an amortization schedule, calculates the interest amount paid over a range of payments.

Command Syntax

`ΣInt(payment1, payment2, [roundvalue])`

Menu Location

On the TI-83, press:

1. 2nd FINANCE to access the finance menu.
2. ALPHA A to select $\Sigma\text{Int}($, or use arrows and ENTER.

On the TI-83+ or higher, press:

1. APPS to access the applications menu.
2. 1 or ENTER to select Finance...
3. ALPHA A to select $\Sigma\text{Int}($, or use arrows and ENTER.

Calculator Compatibility

TI-83/84/+SE

Token Size

2 bytes

Related Commands

- bal(
- Σ Prn(
- tvm_Pmt

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/sigmain>

The Σ Prn(Command

The Σ Prn(command calculates, for an amortization schedule, the principal amount over a range of payments: the portion of those payments that went toward paying off the principal. Its two required arguments are *payment1* and *payment2*, which define the range of payments we're interested in. However, it also uses the values of the finance variables PV, PMT, and I% in its calculations.

The optional argument, *roundvalue*, is the number of digits to which the calculator will round all internal calculations. Since this rounding affects further steps, this isn't the same as using round(to round the result of Σ Prn(to the same number of digits.

Usually, you will know the values of **N**, PV, and I%, but not PMT. This means you'll have to use the finance solver to solve for PMT before calculating Σ Prn(); virtually always, FV will equal 0.

Sample Problem

Imagine that you have taken out a 30-year fixed-rate mortgage. The loan amount is 0000, and the annual interest rate (APR) is 8%. Payments will be made monthly. How much of the principal amount was paid in the first five years?

We know the values of **N**, I%, and PV, though we still need to convert them to monthly values (since payments are made monthly). **N** is 30×12 , and I% is $8/12$. PV is just 100000.

Now, we use the finance solver to solve for PMT. Since you intend to pay out the entire loan, FV is 0. Using either the interactive TVM solver, or the tvm_Pmt command, we get a value of about -3.76 for PMT.

We are ready to use Σ Prn(. We are interested in the payments made during the first five years; that is, between the 1st payment and the $5 \times 12 = 60^{\text{th}}$

```
30*12→N:8/12→I%:  
100000→PV:0→FV  
0  
tvm_Pmt→PMT  
-733.7645739  
 $\Sigma$ Prn(1,60)  
-4930.143283
```

Command Summary

For an amortization schedule, calculates the principal amount paid over a range of payments.

Command Syntax

```
 $\Sigma$ Prn(payment1, payment2,  
[roundvalue])
```

Menu Location

On the TI-83, press:

1. 2nd FINANCE to access the finance menu.
2. 0 to select Σ Prn(, or use arrows and ENTER.

On the TI-83+ or higher, press:

1. APPS to access the applications menu.
2. 1 or ENTER to select Finance...
3. 0 to select Σ Prn(, or use arrows and ENTER.

Calculator Compatibility

TI-83/84/+/SE

Token Size

payment. $\Sigma\text{Prn}(1,60)$ gives us the answer: -30.14 (the negative sign simply indicates the direction of cash flow)

2 bytes

Formulas

The formula that the calculator uses for $\Sigma\text{Prn}($ is in terms of bal(:

$$\Sigma\text{Prn}(n_1, n_2) = \text{bal}(n_2) - \text{bal}(n_1) \quad (1)$$

When the *roundvalue* argument isn't given, we can substitute the explicit formula for bal(and simplify to get the following formula:

$$\Sigma\text{Prn}(n_1, n_2) = \left(\text{PV} - \frac{\text{PMT}}{I\%/100} \right) \left[\left(1 - \frac{I\%}{100} \right)^{n_1} - \left(1 - \frac{I\%}{100} \right)^{n_2} \right] \quad (2)$$

Error Conditions

- ERR:DOMAIN is thrown if either payment number is negative or a decimal.

Related Commands

- bal(
- ΣInt (
- tvm_Pmt

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/sigmaprn>

The $\chi^2\text{cdf}($ Command

$\chi^2\text{cdf}($ is the χ^2 cumulative density function. If some random variable follows a χ^2 distribution, you can use this command to find the probability that this variable will fall in the interval you supply.

The command takes three arguments. *lower* and *upper* define the interval in which you're interested. *df* specifies the degrees of freedom (choosing one of a family of χ^2 distributions).

Advanced Uses

Often, you want to find a "tail probability" - a special case for which the interval has no lower or no upper bound. For example, "what is the probability x is greater than 2?". The TI-83+ has no special symbol for infinity, but you can use E99 to get a very large number that will work equally well in this case (E is the decimal exponent obtained by pressing [2nd]

```
χ²cdf(5, e99, 15  
.9921264114  
χ²cdf(0, 5, 15  
.0078735886  
χ²cdf(0, e99, 15  
1
```

Command Summary

Finds the probability for an interval of the χ^2 distribution.

Command Syntax

$\chi^2(lower, upper, df)$

Menu Location

[EE]). Use E99 for positive infinity, and -E99 for negative infinity.

The $\chi^2\text{cdf(}$ command is crucial to performing a χ^2 goodness of fit test, which the early TI-83 series calculators do not have a command for (the $\chi^2\text{-Test(}$ command performs the χ^2 test of independence, which is not the same thing, although the manual always just refers to it as the "math>" Test"). This test is used to test if an observed frequency distribution differs from the expected, and can be used, for example, to tell if a coin or die is fair.

The Goodness-of-Fit Test routine on the routines page will perform a χ^2 goodness of fit test for you. Or, if you have a TI-84+/SE with OS version 2.30 or higher, you can use the $\chi^2\text{GOF-Test(}$ command.

Press:

1. 2ND DISTR to access the distribution menu
2. 7 to select $\chi^2\text{cdf(}$, or use arrows.

Press 8 instead of 7 on a TI-84+/SE with OS 2.30 or higher.

Calculator Compatibility

TI-83/84/+/SE

Token Size

2 bytes

Formulas

As with other continuous distributions, we can define $\chi^2\text{cdf(}$ in terms of the probability density function:

$$\chi^2\text{cdf}(a, b, k) = \int_a^b \chi^2\text{pdf}(x, k) dx \quad (1)$$

Related Commands

- $\chi^2\text{pdf(}$
- $\text{Shadex}^2($

See Also

- Goodness-of-Fit Test

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/chisquarecdf>

The $\chi^2\text{GOF-Test(}$ Command

The $\chi^2\text{GOF-Test(}$ command performs a χ^2 goodness-of-fit test. Given an expected ideal distribution of a variable across several categories, and a sample from this variable, it tests the hypothesis that the variable actually fits the ideal distribution. As a special case, you could take the ideal distribution to be evenly divided across all categories. Then, the goodness-of-fit test will test the hypothesis that the variable is independent of the category.

The command takes three arguments:

Command Summary

Performs a χ^2 goodness-of-fit test.

Command Syntax

$\chi^2\text{GOF-Test(observed, expected, df)}$

Menu Location

While editing a program, press:

- An *observed* list with an element for each category: the element records the number of times this category appeared in the sample.
- An *expected* list with an element for each category: the element records the frequency with which the category was expected to appear.
- The *degrees of freedom* — usually taken to be one less than the number of categories.

The output is two-fold:

- The test statistic, χ^2 . If the null hypothesis (that the variable fits the distribution) is true, this should be close to 1.
- The probability, p, of the observed distribution assuming the null hypothesis. If this value is low (usually, if it's lower than .05, or lower than .01) this is sufficient evidence to reject the null hypothesis, and conclude that the variable fits a different distribution.

Sample Problem

Working as a sales clerk, you're wondering if the number of customers depends on the day of week. You've taken a count of the number of customers every day for a week: 17 on Monday, 21 on Tuesday, 18 on Wednesday, 10 on Thursday, 24 on Friday, 28 on Saturday, and 24 on Sunday. Store this observed count: {17,21,18,10,24,28,24} to L1.

There were a total of $\text{sum(L1)}=142$ customers. So the expected number of customers on each day was $142/7$. Store all the expected counts: $\{142/7, 142/7, 142/7, 142/7, 142/7, 142/7, 142/7\}$ to L2 (as a shortcut, you can store $142/7\{1,1,1,1,1,1,1\}$).

Since there are 7 days, there are 6 (one less) degrees of freedom. So the resulting command is $\chi^2\text{GOF-Test}(L1, L2, 6)$.

The output will give a χ^2 of 10.32394366, and a p value of 0.1116563376. This is higher than 5%, so the test is not significant on a 95 percent level. It's perfectly possible, in other words, that the number of customers is independent of the day of week.

(Note that in this case, if you suspected the number of customers to be higher on weekends, you could use a more sensitive test for only two categories: [2-SampTTest](#))

Advanced Uses

The $\chi^2\text{GOF-Test}$ command is only available on the TI-84 Plus and TI-84 Plus SE. However, it's possible to use the $\chi^2\text{cdf}$ command to simulate it on the other calculators: see the [\$\chi^2\$ Goodness-of-fit Test](#) routine.

Formulas

The formula for calculating the test statistic is as follows (O_i is the observed count of the i^{th} category, and E_i is the expected count):

1. STAT to access the statistics menu.
2. LEFT to access the tests submenu.
3. ALPHA D to select $\chi^2\text{GOF-Test}()$.

(outside the program editor, this will select the $\chi^2\text{GOF-Test... interactive solver}$)

Calculator Compatibility

TI-84+/SE, OS 2.30 or higher

Token Size

2 bytes

$$\chi^2_{n-1} = \sum_{i=1}^n \frac{(O_i - E_i)^2}{E_i}$$

The p-value, then, is the probability that the χ^2 statistic would be this high, using the $\chi^2\text{cdf}$ command with the appropriate value for degrees of freedom.

Error Conditions

- ERR:DIM MISMATCH is thrown if the two lists are of different length.
- ERR:DOMAIN is thrown if they only have one element, or if df is not a positive integer.

Related Commands

- $\chi^2\text{-Test}$

See Also

- χ^2 Goodness-of-fit Test

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/chisquaregof-test>

The $\chi^2\text{pdf}$ (Command

$\chi^2\text{pdf}$ (is the χ^2 probability density function.

Since the χ^2 distribution is continuous, the value of $\chi^2\text{pdf}$ (doesn't represent an actual probability — in fact, one of the only uses for this command is to draw a graph of the χ^2 curve. You could also use it for various calculus purposes, such as finding inflection points.

The command takes two arguments: the value at which to evaluate the p.d.f., and df , the number of 'degrees of freedom'.

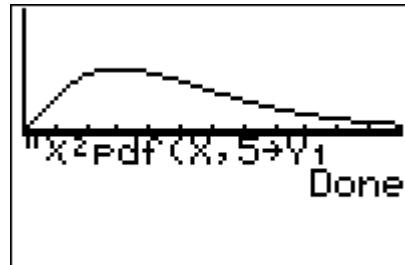
Formulas

The value of $\chi^2\text{pdf}$ (is given by

$$\chi^2\text{pdf}(x, k) = \frac{(1/2)^{k/2}}{(k/2 - 1)!} x^{k/2-1} e^{-x/2} \quad (1)$$

Related Commands

- $\chi^2\text{cdf}$
- Shadex(



Command Summary

Evaluates the χ^2 probability density function at a point.

Command Syntax

$\chi^2\text{pdf}(x, df)$

Menu Location

Press:

1. 2ND DISTR to access the distribution menu
2. 6 to select $\chi^2\text{pdf}$ (, or use arrows.

Press 7 instead of 6 on a TI-84+/SE

with OS 2.30 or higher.

Calculator Compatibility

TI-83/84/+SE

Token Size

2 bytes

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/chisquarepdf>

The χ^2 -Test(Command

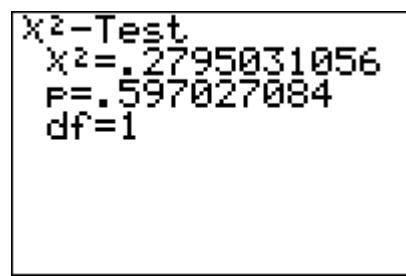
This command performs a χ^2 test of independence. This test is used to assess the independence of two categorical variables with known frequencies. The test is only valid for a simple random sample from the population, and only if all the frequencies are sufficiently large (greater than 5).

Note: this test is different from the χ^2 goodness of fit test, which the TI-83 calculators don't have a command form. For a program that will perform the χ^2 goodness-of-fit test, see the [goodness-of-fit test](#) routine.

To use this test, you need a [matrix](#) containing a *contingency table*. This is a table in which every row corresponds to a value of the first variable, and every column to a value of the second. The number in each cell represents the frequency with which the corresponding values of the two variables occur together. For example: suppose that the two variables are sex (male and female) and eye color (blue, brown, and green). The contingency table would have two rows and three columns. The cell in the first row and column would be the number of blue-eyed men in the sample, the cell in the second row and first column would be the number of blue-eyed women, and so on.

The χ^2 -Test(command takes two arguments: the *observed* matrix and *expected* matrix. The first of these should be the contingency table you've already completed, presumably in the Matrix editor. The expected matrix does not need to already exist: the χ^2 -Test(command will calculate and store the expected frequencies (under the assumption that the variables are independent) to this matrix.

The command is primarily for use in a program. Although you can access the χ^2 -Test(command on



X²-Test
X²=.2795031056
P=.597027084
df=1

Command Summary

Performs a χ^2 test of independence.

Command Syntax

$\chi^2\text{-Test}(\text{observed matrix}, \text{expected matrix}, \text{draw?})$

Menu Location

While editing a program, press:

1. STAT to access the statistics menu
2. LEFT to access the TESTS submenu
3. ALPHA C to select χ^2 -Test(, or use arrows

(outside the program editor, this will select the χ^2 -Test... interactive solver)

Calculator Compatibility

TI-83/84/+SE

Token Size

2 bytes

the home screen, via the catalog, there's no need:
you can use the χ^2 -Test... interactive solver found in
the menu instead.

↳ [Styles](#)

In either case, it's important to understand the output of χ^2 -Test(). Here are the meanings of each line:

- χ^2 is the test statistic, calculated from the differences between the observed and the expected matrices.
- p is the probability associated with the test statistic. We use p to test the null hypothesis that the two variables are independent. If p is low (usually, if it's <0.05) this means there's little chance that two independent variables would have a contingency table so different from the expected, and we reject the null hypothesis (so we'd conclude that the two variables are not independent).
- df is the degrees of freedom, defined as (# of rows - 1)*(# of columns - 1), important for calculating p.

Sample Problem

You want to compare the effectiveness of three treatments in curing a terminal disease. You have obtained data for 100 patients who had the disease, which contained information on the treatment used, and whether the patient lived or died. You put this information in a contingency table:

	Lived	Died
Treatment A	40	10
Treatment B	27	6
Treatment C	11	6

To perform the test, you store this information to a matrix such as [A], either through the matrix editor or by hand:

```
: [[40,10],[27,6],[11,6→[A]
```

You submit this matrix as the first argument, and some other matrix (such as [B]) for the second:

```
: χ²-Test([A],[B]
```

The output looks something like this:

```
χ²-Test
χ²=2.14776311
p=.3416796916
df=2
```

The most important part of this output is the line p=.3416796916 - the probability of getting such results under the hypothesis that the treatments and survival rate are independent. This value is greater than .05, so the data is not significant on a 5% level. There is not enough evidence to reject the null hypothesis, so treatment and survival rate may very well be independent. In non-

mathematical language, this means that there's no reason to believe the treatments vary in effectiveness.

Advanced Uses

The final argument of χ^2 -Test(, *draw?*, will display the results in a graphical manner if you put in "1" for it. The calculator will draw the χ^2 distribution with the correct degrees of freedom, and shade the area of the graph beyond the χ^2 statistic. In addition, the same values as usually will be calculated and displayed. You would make your conclusions in the same way as for the regular output.

Related Commands

- [\$\chi^2\$ GOF-Test\(](#)
- [\$\chi^2\$ cdf\(](#)

See Also

- [Goodness-of-Fit Test](#)

For the most up-to-date version of this command, see <http://tibasicdev.wikidot.com/chisquare-test>

Blinking Text

```
:length(Str1→N:1  
:Repeat getKey  
:If dim(rand(15  
:Output(A,B,sub(Str1+" (N spaces) ",  
:not(Ans  
:End
```



By leaving 1 by itself on one line, we store it to *Ans*, which will be easier to work with. Then, the Repeat getKey loop will keep blinking the text until a key — any key — is pressed.

Output(A,B,sub(Str1+" (N spaces) ",1+NAns,N will display either the text or the equivalent number of spaces on the screen at coordinates (A,B). If you want to blink the text "Hello", for example, then you would need to use five spaces. We negate *Ans*'s value with not(, which acts as a flag to control the blinking.

If *dim(rand(15* is a clever way of delaying the blinking, so that it doesn't blink too fast. *rand(15* generates a list of 15 random numbers, which is a slightly time-consuming process. If *dim(* is just a way of wrapping this list so it doesn't change *Ans*. Since *dim(rand(15* is always 15, the *If* statement will

Routine Summary

Creates a blinking effect on the home screen.

Inputs

Str1 - the text to blink on the screen
A,B - the Output(coordinates for the text
N - the length of the text to be displayed

Outputs

None

Variables Used

Str1, N, A, B, Ans

Calculator Compatibility

TI-83/84/+/SE

Download

...

always be true, so we don't have to worry about the next line being skipped. By changing 15 to a lower or higher number, you can make the blinking go faster or slower, respectively.

For the most up-to-date version of this routine, see <http://tibasicdev.wikidot.com/blinking-text>

[BLINK.zip](#)

Day of Week

```
: round(7fPart(dbd(101.5, DE2+M+fPart(
```

Using the dbd(command, we return the number of days between the given date and January 1, 1950, our base date. dbd('s argument, in this case, is of the form DDMM.YY, so we put the date we're given in that form with DE2+M+fPart(.01Y. Once we have the number of days between the two dates, we divide it by seven (as there are seven days in a week) to get the number of weeks.

However, we are not interested in the number of weeks between two dates, we are interested in how far into the week we are. That would be the fractional part of the number of weeks. For example, if there was fourteen days between two dates (the first date being a Sunday) and we divide that by seven, we would have the equation $14/7=2.00$. The .00 means that the week hasn't yet begun (and so, the day is Sunday).

If there was 15 days between two dates, the equation would give $15/7=2.1428571$. The .1428571 means that we are .1428571 into the week, and if we multiply that by seven, we get the equation $.1428571*7=1$. This means that we are one day into the week (and so, the day is Monday). So in our formula, after finding out the number of days between two dates, we find the fractional part of it, and multiply that by seven. Finally, round(gets rid of any minor rounding errors.

As the 1st of January, 1950 was a Sunday (which is the date we have inputted into the above routine), so the number of days into the week will always be relative to Sunday. That is, the output 0 through 6 is relative to the days Sunday through Saturday respectively. If you want 0 to be Monday instead, make the base date January 2 instead by changing 101.5 to 201.5.

This routine can handle dates from January 1, 1950 to December 31, 2049. For other dates, it will assume the year is in that range instead.

Error Conditions

- ERR:DOMAIN if the date is invalid.

Routine Summary

Calculates the day of week of a date, without using dayOfWk (because it's only available on the 84+ and 84+SE)

Inputs

D - The day

M - The month

Y - The year (1950 through 2049)

Outputs

Ans - 0-6 = the day of week
(Sunday - Saturday)

Variables Used

D, M, Y, Ans

Calculator Compatibility

TI-83/84/+/SE

Download

[weekday.zip](#)

Decimal to Fraction

```
:Ans→X:{1,abs(Ans  
:Repeat E-9>Ans(2  
:abs(Ans(2){1,fPart(Ans(1)/Ans(2  
:End  
:iPart({X,1}/Ans(1
```

Although there is a ►Frac command available for converting a decimal number to a fraction, this is only a formatting command and doesn't actually give you the numerator and denominator as separate numbers. This limits your choices for using fractions, especially since ►Frac only works with the Disp and Pause commands. In addition, it does not work very well, and fails for several inputs that you would think are within its ability to figure out (such as -1.3427625). Fortunately, you can improve upon the ►Frac command by using your own routine.

The basic algorithm that you use when converting a number to a fraction is commonly known as the Euclidean algorithm. While it has been around for literally millennia, it is still one of the best algorithms because of its sheer simplicity (it doesn't require any factoring or other complex math operations).

The way it works is that you have two numbers (in our routine, it's one and the decimal number you input), and you divide the first number by the second number, and check to see if there is a remainder. If there is a remainder, you then repeat the process again, except this time you swap the numbers. This process is repeated over and over again until the second number is equal to zero, at which point you will have your desired fraction.

One thing you will probably notice is that we aren't actually checking for zero in the Repeat loop. Because of the way that TI designed the TI-Basic language, each number has a limited amount of precision. As a result, any math calculations that involve extremely small or large numbers will produce rounding errors that don't return the right number. The easiest way to deal with this problem is by checking for a really small number (in this case, E⁻⁹).

For the most up-to-date version of this routine, see <http://tibasicdev.wikidot.com/decimal-to-fraction>

Routine Summary

Converts a decimal number to a fraction.

Inputs

Ans - the number you want to convert to a fraction

Outputs

Ans - the fraction in list format

Variables Used

Ans, X

Calculator Compatibility

TI-83/84/+SE

Author

Weregoose

URL: [United TI](#)

Download

[decimaltofraction.zip](#)

Deck of Cards

Creating the deck

```
seq(X/4,X,4,55→DECK
```

Shuffling the deck

```
rand(52→L1
SortA(L1,DECK
```

Accessing individual cards

```
LDECK(I
Disp "VALUE:", sub("A23456789TJQK", i
Disp "SUIT:", sub("SHCD", 4fPart(Ans)+
```

The cards in the deck are stored in the form Value.Suit, where the value of the card (Ace through King) is encoded as a number 1 through 13; and the suit of the card is the fractional part, encoded as one of 0, 0.25, 0.5, or 0.75. In the above code (accessing individual cards), the convention is that 0=Spades, 0.25=Hearts, 0.5=Clubs, 0.75=Diamonds; however, you can pick any order as long as you're consistent.

Since any value from 1.00 to 13.75, that's 1/4 of an integer, is a valid card, we can generate the entire deck as 1/4 of the values {4,5, ..., 54, 55}.

When shuffling the deck, we generate a random list in L₁, then use SortA(to sort L_{DECK} by the values in L₁. Since the values in L₁ are random, this has the effect of sorting L_{DECK} in a random order.

The main overhead of this shuffling method, however, is that generating a random list might take a long time (around a second or two). To avoid this, you can generate individual elements of L₁ randomly in a getKey loop, while waiting for a key, then use L₁ to shuffle later. Since shuffling isn't done often, by the time you need to shuffle, L₁ will most likely be fully randomized already.

Finally, accessing the cards is done using fPart(and int(. If a variable X is encoded in the same way that we encode cards, int(X) will return its value (1-13) and 4fPart(X)+1 will return its suit as a number 1-4.

For the most up-to-date version of this routine, see <http://tibasicdev.wikidot.com/deck-of-cards>

Routine Summary

Simulates a standard 52-card deck of cards.

Inputs

None

Outputs

L_{DECK} - the cards, in the format:

Value.Suit

Value is 1..13
Suit is 0, .25, .5, .75

Variables Used

L₁, L_{DECK}

Calculator Compatibility

TI-83/84/+/SE

Download

[deckofcards.zip](#)

Draw Ellipse

Setup:

```
:cos(.1πcumSum(binomcdf(20,0→X
:sin(.1πcumSum(binomcdf(20,0→Y
```

Routine Summary

Draws an ellipse with specified

```
:Plot1(xyLine,LX,LY,(dot)
:PlotsOff
```

Main code:

```
:StorePic 1
(:ZoomSto)
:- (B+D)/abs(D-B->Xmin
:(A+C-124)/abs(C-A->Ymin
:2/abs(D-B->ΔX
:2/abs(C-A->ΔY
:PlotsOn 1
:RecallPic 1
:StorePic 1
:PlotsOff
(:ZoomRcl)
:RecallPic 1
```

To use the routine, add the setup code to the beginning of your program. Then, to draw an ellipse, initialize (A,B) and (C,D) to be opposite corners of an imagined rectangle, in pixel coordinates, and run the main code of the routine. The ellipse will be drawn inside this rectangle (much like the functionality of the circle tool in Paint). Unlike built-in pixel commands, this routine doesn't require the pixels to be on-screen: you can use negative coordinates, and coordinates past the 62nd row and 94th column - of course, if you do, then the off-screen part of the ellipse won't be drawn.

As for speed, the routine is far faster than the Circle(command. If you use Circle(with its undocumented fourth argument, that is a bit faster still - but doesn't allow you as much control over the shape of the ellipse as this routine does.

This routine draws an ellipse given its pixel coordinates in two overall steps:

1. First, it calculates the window dimensions so that the unit circle (centered at the origin, with radius 1) will be stretched to the required pixel coordinates.
2. Next, it draws the unit circle.

If the unit circle is stretched to fit in the rectangle from (A,B) to (C,D) then we know the following:

- The vertical distance from A to C (in pixel rows) should be equivalent to the window distance 2 (from -1 to 1). This allows us to solve for ΔY .
- The horizontal distance from B to D (in pixel columns) should also be equivalent to the window distance 2. This allows us to solve for ΔX .
- The pixel most nearly corresponding to the midpoint between (A,B) and (C,D) should be equivalent to the point (0,0), the center of the circle. Given ΔX and ΔY , this allows us to solve for X_{min} and Y_{min} .

The exact math isn't significant, but it gives rise to the formulas you see in the routine. Note that by using the abs(command, we ensure that the order of the two points (A,B) and (C,D) isn't important: you can switch A and C, or B and D, and still get the same circle.

The code for actually drawing the circle uses a look-up table for sine and cosine (which defines

coordinates.

Inputs

(A,B) - upper left corner, in pixels
(C,D) - lower right corner, in pixels

Outputs

Ellipse inscribed in the rectangle from (A,B) to (C,D)

Variables Used

A,B,C,D for input
LX and LY store a sin/cos look-up table
Plot1 to store the unit circle
Pic1 to preserve background

Calculator Compatibility

All

Author

Mikhail Lavrov (DarkerLine)

URL: <http://mpl.unitedti.org/?p=13>

20 points spaced around the circle). The table is calculated in the setup code, and then an xyLine plot is initialized, which will connect these 20 points when graphed. Now, to draw the unit circle, all that needs to be done is to enable the plot, using the PlotsOn command. This actually draws a 20-gon, rather than a circle, but the difference is imperceptible even on large circles.

The rest of the code is necessary to preserve the graph screen as the window changes and Plot1 is enabled and disabled (which causes the screen to be cleared), using StorePic and RecallPic. Optionally, you can also preserve the window dimensions, using ZoomSto and ZoomRcl — this is useful if your program mixes pixel and point commands.

The routine uses three fairly large variables to do its work: LX , LY , and Pic1 . LX and LY need to be preserved from the setup code to whenever the routine is called; Pic1 is only used temporarily. It's a good idea to clean up all three by deleting them at the end of the program.

For the most up-to-date version of this routine, see <http://tibasicdev.wikidot.com/draw-ellipse>

Highlighting Text

```
:Text(A,B,Str1  
:For(A,A,A+6  
:For(C,B-1,B+W  
:Px1-Change(A,C  
:End:End
```

First, we display the string using Text(on the graph screen. This displayed it normally, using black on white. To switch it to white on black, we need to flip the pixels from white to black and black to white — this can be done using Pxl-Change(. We chose the pixel command over the point command for two reasons: first, it is slightly faster, and second, it uses the same coordinate system as Text(— pixels (if we used Pt-Change, we'd need to convert from pixels to points to be sure of drawing in the same location).

We loop A from above to below the text, and C from just before the text to immediately after it. Note that because of the way the For(loop works — it saves the lower and upper bound before doing the loop — we can reuse the variable A. But we can't reuse B in the same way in the second For(loop, because the loop is done multiple times — it would work correctly the first time, but then B would have changed for the second.

Since Text(uses variable-width font, we need the W variable to tell us how long the string is in pixels. For uppercase letters, the width is 4 pixels; for spaces, the width is 1 pixel, and for lowercase letters, the width varies from 2 to 6 pixels.

Because Pxl-Change(must not go off the screen, check to see that the string fits on the screen entirely (with a border of 1 pixel around it) before displaying it.

Routine Summary

Highlights text, displaying it in white on a black background.

Inputs

Str1 - text to be displayed
A - row coordinate of the text
B - column coordinate of the text
W - pixel width of the text

Outputs

None

Variables Used

Str1, A, B, C, W

Calculator Compatibility

TI-83/84/+/SE

Download

[highlighttext.zip](#)

Error Conditions

- **ERR:DOMAIN** is thrown if the string does not fit on the screen entirely.

Related Routines

- [Wordwrapping Text on the Graphscreen](#)

For the most up-to-date version of this routine, see <http://tibasicdev.wikidot.com/highlighting-text>

Key Code Retriever

```
:Repeat Ans  
:getKey  
:End  
:Disp Ans
```

This routine loops until a key is pressed. When a key is pressed, the key code for that specific key is displayed.

Routine Summary

Retrieves the key code of any key pressed

Calculator Compatibility

TI-83/84/+/SE

For the most up-to-date version of this routine, see <http://tibasicdev.wikidot.com/key-code-retriever>

List Frequency

```
:{L1(1→L2  
:{1→L3  
:For(I,2,dim(L1  
:If max(L2=L1(I:Then  
:L3+(L2=L1(I→L2  
:Else  
:augment(L2, {L1(I→L2  
:augment(L2, {1→L3  
:End:End
```

With our list of values stored in L₁, we store the first element of L₁ to L₂ to initialize it, and store a value of one to L₃ because that value has appeared once in the list (by default it is the first value and thus we know it appeared). We then begin looping in the second element to avoid an extra increment and then through all of the elements of L₁.

When determining the frequency of a value, we need to check to see if the particular element (in this

Routine Summary

Returns a list of the frequency of values in another list.

Inputs

L₁ - the list you want to find the frequency of

Outputs

L₂ - the values of L₁ without repetition

L₃ - the frequencies of the values in the list L₂

Variables Used

L₁, L₂, L₃, I

case, $L_1(I)$ has already appeared in L_2 . If it has, we increment (add one) to its respective frequency in L_2 . However, if the value has never appeared in L_1 , we add that value as a new element to the end of L_2 and also set its frequency to one. We repeat this until we have the frequency of all of the values in L_1 stored in L_2 .

When you are done using L_1 , L_2 , and L_3 , you should clean them up at the end of your program.

Calculator Compatibility

TI-83/84/+/SE

Author

DarkerLine

URL: [United TI](#)

Download

[listfrequency.zip](#)

For the most up-to-date version of this routine, see <http://tibasicdev.wikidot.com/list-frequency>

List to String

```
: "?  
: For(A,1,dim(L1)  
: Ans+sub("ABCDEFGHIJKLMNPQRSTUVWXYZ  
: End  
: sub(Ans,2,length(Ans)-1→Str1
```

With our list of values stored in L_1 , we loop through each element of L_1 and store the character to our string that is at the matching position in our substring. In order to construct a string with all of the characters from L_1 , we first need to create a dummy string. This is what the "?" is used for.

Each time through the For(loop, we concatenate the string from before (which is still stored in the Ans variable) to the next character that is found in the list. Using Ans allows us to not have to use another string variable, since Ans can act like a string and it gets updated accordingly, and Ans is also faster than a string variable.

By the time we are done with the For(loop, all of our characters are put together in Ans. However, because we stored the dummy character as the first character at the beginning of the string, we now need to remove it, which we do by simply getting the substring from the second character to the end of the string. Finally, we store the string to a more permanent variable (in this case, Str1) for future use.

This routine only allows for values from 1 to 26 in the list, since our string of characters is the uppercase alphabet, and each list value must match up to one of the string positions. If you add more characters to the string, however, you can increase the range of values in the list. This routine uses L_1 , so you should clean it up at the end of your program.

Routine Summary

Converts a list of numbers to a string.

Inputs

L_1 - The list you want to convert

Outputs

$Str1$ - The string that the text is stored to

Variables Used

L_1 , A , Ans , $Str1$

Calculator Compatibility

TI-83/84/+/SE

Download

[listtostring.zip](#)

Related Routines

- [Number to String](#)
- [Matrix to String](#)

For the most up-to-date version of this routine, see <http://tibasicdev.wikidot.com/list-to-string>

Marquee

```
:Str1
:Repeat getKey
:Output(A,B,sub(Ans,1,N
:sub(Ans,2,length(Ans)-1)+sub(Ans,1,
:If dim(rand(4
:End
```



By leaving Str1 by itself on one line, we store it to Ans, which will be easier to work with. Then, the Repeat getKey loop will display the marquee until a key — any key — is pressed.

Output(A,B,sub(Ans,1,N will display N characters of Str1 at A,B. Then, the next line will rotate Str1, so that the next time we're at this point in the loop, the string shifts one character.

Finally, If dim(rand(4 is a clever way of delaying the marquee, so it doesn't scroll too fast. rand(4 generates a list of 4 random numbers, which is a slightly time-consuming process. If dim(is just a way of wrapping this list so it doesn't change Ans. Since dim(rand(4 is always 4, the If statement will always be true, so we don't have to worry about the next line being skipped. By changing 4 to a lower or higher number, you can make the marquee go faster or slower, respectively.

Error Conditions

- **ERR:INVALID DIM** is thrown if the length N is longer than the number of characters in Str1.

Routine Summary

Scrolls a string across one line, in marquee fashion, on the home screen.

Inputs

Str1 - the text to be scrolled
A,B - the Output(coordinates for the text
N - the number of characters to display at a time

Outputs

None

Variables Used

Str1, A, B, N, Ans

Calculator Compatibility

TI-83/84/+/SE

Author

Weregoose

URL: [United TI](#)

Download

[marquee.zip](#)

For the most up-to-date version of this routine, see <http://tibasicdev.wikidot.com/marquee>

Matrix to String

```
:dim([A]→L1:"?
:For(A,1,L1(1
:For(B,1,L1(2
:Ans+sub("ABCDEFGHIJKLMNPQRSTUVWXYZ
:End:End
:sub(Ans,2,length(Ans)-1→Str1
```

With our values stored in [A], we get the dimensions (row x column) of the matrix, which are stored in a list. We then loop through each row and column, and store the character at the respective element to our string that is at the matching position in our substring. In order to construct a string with all of the characters from [A], we first need to create a dummy string. This is what the "?" is used for.

Each time through the For(loops, we concatenate the string from before (which is still stored in the Ans variable) to the next character that is found in the matrix. Using Ans allows us to not have to use another string variable, since Ans can act like a string and it gets updated accordingly, and Ans is also faster than a string variable.

By the time we are done with the For(loops, all of our characters are put together in Ans. However, because we stored the dummy character as the first character at the beginning of the string, we now need to remove it, which we do by simply getting the substring from the second character to the end of the string. Finally, we store the string to a more permanent variable (in this case, Str1) for future use.

This routine only allows for values from 1 to 26 in the matrix, since our string of characters is the uppercase alphabet, and each matrix value must match up to one of the string positions. If you add more characters to the string, however, you can increase the range of values in the matrix. This routine uses [A] and L₁, so you should clean them up at the end of your program.

Related Routines

- [Number to String](#)
- [List to String](#)

For the most up-to-date version of this routine, see <http://tibasicdev.wikidot.com/matrix-to-string>

Number Concatenation

```
:N+M10^(1+int(log(N→0
```

With our two numbers stored in M and N respectively, we add the first number to the second number by raising it to the 10th power, with the

Routine Summary

Converts a matrix to a string.

Inputs

[A] - The matrix you want to convert

Outputs

Str1 - The string that the text is stored to

Variables Used

[A], L₁, A, B, Ans, Str1

Calculator Compatibility

TI-83/84/+/SE

Download

[matrixtoString.zip](#)

Routine Summary

Concatenates two whole numbers together.

exponent being how many digits are in the number. We then store this result to a new variable for later use.

We can figure out how many digits are in a number by using the `1+int(log(` trick. This trick works with any positive whole numbers, and if you add `abs(` after `log(`, it will also work with negative numbers. Unfortunately, it does not work with decimals.

Inputs

`M` - the first number
`N` - the second number

Outputs

`O` - the concatenated number

Variables Used

`M, N, O`

Calculator Compatibility

TI-83/84/+/SE

Author

DarkerLine

URL: [United TI](#)

Download

[numberconcatenation.zip](#)

For the most up-to-date version of this routine, see <http://tibasicdev.wikidot.com/number-concatenation>

Number Subset

```
:10^(2-B+int(log(A
:int((A-int(A/Ans)Ans)/Ans10^(C
```

With our number stored in `A`, and the starting position and length of the subset stored in `B` and `C` respectively, we get the subset of the number by first subtracting the number divided by 10 to the power of `2-B+int(log(A` (which is used to get how many digits are in the number), and then dividing that result by multiplying 10 to the power of `2-B+int(log(A` and 10 to the power of `C` (which is the length of the subset).

A simple example should help you understand this routine. Say you input the number 123, with a starting position of 2 and a length of 2, it will return a result of 23. You can also use negative and decimal numbers with the routine, and it will still work correctly: a number of 13579.02468 with a starting position of 4 and length 4 will return 7902.

This routine is comparable to the `sub(` command

Routine Summary

Returns a subset of a number.

Inputs

`A` - the number to get the subset from
`B` - the starting position of the subset
`C` - the length of the subset

Outputs

`Ans` - the subset of the number

Variables Used

`A, B, C, Ans`

Calculator Compatibility

TI-83/84/+/SE

that works with [strings](#).

TI-83/84/+SE

Author

Wegoose

URL: [United TI](#)

Download

[numbersubset.zip](#)

For the most up-to-date version of this routine, see <http://tibasicdev.wikidot.com/number-subset>

Number to String

```
: {0, 1→L1
: {0, N→L2
: LinReg(ax+b) Y1
: Equ►String(Y1, Str1
: sub(Str1, 1, length(Str1)-3→Str1
```

This code works because it creates two points with a known best fit line: the best fit line through (0,0) and (1,N) is $y=Nx+0$. [LinReg\(ax+b\)](#) calculates this best fit line, and stores its equation to Y_1 .

Then, we use [Equ►String\(](#) to store this equation to $Str1$, which now contains "NX+0" with N replaced by the numerical value of N . After that, the [sub\(](#) command gets rid of the "X+0" at the end, leaving only the string representation of N .

This routine uses L_1 , L_2 , and Y_1 , so you should [clean up](#) those variables at the end of your program. If you're working with the graph screen in [function](#) mode, storing to Y_1 can be a problem since it will draw an unwanted line through your graphics. Use r_1 instead but make sure the calculator isn't in [polar](#) mode.

Note: This only works for real numbers. With complex numbers, such as imaginary numbers, you can use this code at the end of the first to get the same effect with i in it.

```
: Str1+"i"→Str1
```

Routine Summary

Converts a real number to a string.

Inputs

N - the number you want to convert

Outputs

$Str1$ - the number N in string form

Variables Used

L_1 , L_2 , Y_1 , $Str1$, N

Calculator Compatibility

TI-83/84/+SE

Download

[numbertostring.zip](#)

Related Routines

- [List to String](#)
- [Matrix to String](#)

Pad a String

```
:For(X,1,N  
:" "+Ans+"  
:End  
:Ans→Str1
```

With our string stored in Ans, we concatenate (add) a space to the beginning and end of the string. At the same time, the new string is stored to Ans, which is what we use next time through the For(loop. The loop gets repeated over and over again until we have added however many spaces we wanted. If you only want spaces on one side of the string, edit the second line accordingly.

Using Ans allows us to not have to use another string variable, since Ans can act like a string and it gets updated accordingly, and Ans is also faster than a string variable.

We store the string to a more permanent variable (in this case, Str1) for future use. When you are done using Str1, you should clean it up at the end of your program.

Related Routines

- [Strip a String](#)
- [Scramble a String](#)

For the most up-to-date version of this routine, see <http://tibasicdev.wikidot.com/pad-a-string>

Repeat a String

```
:For(X,1,N  
:Ans+Ans→Str1  
:End
```

With our string stored in Ans, we concatenate (add) another copy to the end of the string, and store the new string to Str1. At the same time, the new string is also stored to Ans, which is what we use next time through the loop. The loop gets repeated over and over again until we have added however many

Routine Summary

Pad a string with spaces on the left and right.

Inputs

Ans - The string you want to pad
N - How many spaces you want

Outputs

Str1 - The padded string

Variables Used

X, N, Ans, Str1

Calculator Compatibility

TI-83/84/+SE

Download

[padstring.zip](#)

Routine Summary

Repeats a string however many times you want.

Inputs

Ans - The string you want to repeat
 2^N - How many times you want the string repeated

copies we wanted. If you want to add more than one copy of the string each time through the loop, edit the second line accordingly.

Using Ans allows us to not have to use another string variable, since Ans can act like a string and it gets updated accordingly, and Ans is also faster than a string variable. In addition, by storing the string to Str1 inside the loop, we avoid having to add an additional storage to the routine.

We store the string to a more permanent variable (in this case, Str1) for future use. When you are done using Str1, you should clean it up at the end of your program.

Related Routines

- [Strip a String](#)
- [Pad a String](#)

For the most up-to-date version of this routine, see <http://tibasicdev.wikidot.com/repeat-a-string>

Reverse a String

```
:Str1  
:For(I,1,length(Ans)-1  
:sub(Ans,2I,1)+Ans  
:End  
:sub(Ans,1,I→Str1
```

With our string stored in Str1 and Ans, we loop through each character, starting from the beginning to the end, and add it to the beginning of the string, building the reversed string up at the beginning as we go:

```
12345      (original string - the first character  
212345    (add then second character  
3212345   (continue adding characters  
543212345 (this is what the end result is)
```

Since adding to the beginning of the string alters the indices, we must take that into account — this is where the $2I$ in the formula comes from. It adds the 2nd character the first time, the 4th character (which was originally the 3rd) next, the 6th character (originally the 4th) after that, and so on.

Using Ans allows us to not have to use another string variable, since Ans can act like a string

String repeated

Outputs

Str1 - The repeated string

Variables Used

X, N, Ans, Str1

Calculator Compatibility

TI-83/84/+SE

Download

[repeatstring.zip](#)

Routine Summary

Reverses the characters in a string.

Inputs

Str1 - The string you want to reverse

Outputs

Str1 - The reversed string

Variables Used

Str1, I, Ans

Calculator Compatibility

TI-83/84/+SE

Download

[reversestring.zip](#)

and it gets updated accordingly, and Ans is also faster than a string variable.

By the time we are done with the For(loop, all of our characters are put together in Ans in reverse order, before the original string. To finish, we take the first (reversed) half as a substring and store it back in Str1 for further use. We can use I for this purpose because it looped from 1 to length(Str1)-1, so its value will be length(Str1) when exiting.

If you want to preserve the original string, store it to a different string variable in the first line of the code.

When you are done using Str1, you should clean it up at the end of your program.

For the most up-to-date version of this routine, see <http://tibasicdev.wikidot.com/reverse-a-string>

Run-Length Encoding (RLE) Compression

```
:1→J
:For(I,2,dim(L₁
:If L₁(I)=L₁(J:Then
:.002+L₁(J→L₁(J
:Else
:If L₁(I)=int(L₁(J:Then
:.001+L₁(J→L₁(J
:Else
:J+1→J
:L₁(I→L₁(J
:End:End:End
:J→dim(L₁
```

Run-length encoding (RLE) is a very easy compression algorithm that you can use for compressing a list of numbers. The way it works is that you remove all of the consecutive repeated numbers from the list, and modify the first instance of the numbers with how many repeated numbers there were.

For example, say you have a list of numbers 1,2,2,3,3,3,4. You start with the 1, and since there is only one 1, it wouldn't be modified. There are two 2's, however, so you would remove the second two, and add a decimal part (using fPart()) of how many 2's there were (in this case, just two, which we represent as .002). You would do this for the rest of the list, and the final list would be 1,2.002,3.003,4.

To save memory (which is of course the reason we're compressing) we will store the result to L₁, the same list the uncompressed data is in. Throughout the loop, J is the index of the last element of the compressed part of the list, and I is the index in the uncompressed part. We don't have to worry about the indices colliding, since I is always bigger than J.

Routine Summary

Compresses a list of numbers using RLE compression.

Inputs

L₁ - The list of numbers you want to compress

Outputs

L₁ - The list of compressed numbers

Variables Used

L₁, I, J

Calculator Compatibility

TI-83/84/+/SE

Download

[rlecompress.zip](#)

We loop over the list with I , and check if the current element has the same value as the last element of the compressed list. If it is, then it's the beginning of a run, so we add .002 to that last element.

If it isn't there's another possibility — the last element could represent an existing run of the same element. We check for this with the code $\text{If } L_1(I) = \text{int}(L_1(J))$. If this turns out to be the case, we add .001 to increase the length of the run. Otherwise, the element really is different, and we increase J and add a new element.

At the end, we store J to the size of L_1 . This gets rid of all the unnecessary data, leaving us only with the compressed portion of the list.

Note that we never store anything to L_1 itself, only to its elements. This is a useful technique to avoid using too much memory while the program is running: if we stored to L_1 , a copy of the list would be stored to Ans , which could easily give a ERR:MEMORY if the list is too large. As the program is now, the only additional memory used is contained in three real variables (I , J , and Ans).

Related Routines

- RLE Decompression

For the most up-to-date version of this routine, see <http://tibasicdev.wikidot.com/rle-compress>

Run-Length Encoding (RLE) Decompression

```
:dim(L1)→J  
:sum(E3fPart(L1)+not(fPart(L1→dim(L1)))  
:For(I,Ans,1,-1  
:int(L1(J→L1(I  
:If .001<fPart(L1(J):Then  
:L1(J)-.001→L1(J  
:Else  
:J-1→J  
:End:End
```

Run-length encoding (RLE) is a very easy compression algorithm that you can use for compressing a list of numbers. The way it works is that you remove all of the consecutive repeated numbers from the list, and modify the first instance of the numbers with how many repeated numbers there were.

For example, say you have a list of numbers 1,2,2,3,3,3,4. You start with the 1, and since there is only one 1, it wouldn't be modified. There are two 2's, however, so you would remove the second two, and add a decimal part (using fPart() of how many

Routine Summary

Decompresses a run length-encoded list.

Inputs

L_1 - The compressed list

Outputs

L_1 - The decompressed list

Variables Used

L_1 , I , J

Calculator Compatibility

TI-83/84/+/SE

Download

[rledecompress.zip](#)

2's there were (in this case, just two, which we represent as .002). You would do this for the rest of the list, and the final list would be 1,2.002,3.003,4.

This routine could loosely be described as the RLE compression routine, but backwards. We start by calculating the length of the decompressed list. This is the sum of the length of the runs — $E3fPart(L_1)$ — plus the number of elements with no runs — $\text{not}(fPart(L_1))$. Here E represents the scientific E.

Then, the decompression begins. The routine keeps the following loop invariants (things that stay true after each iteration of the loop):

- Every element after the Ith element is the correct decompressed element in that spot.
- The portion of the list up to and including the Jth element is the compressed version of the list elements that will be 1 through I.

We "unpack" one element from the end of the compressed portion: $\text{int}(L_1(J \rightarrow L_1(I$. Then we test if this compressed portion is a run that still contains more elements. If it is, we subtract .001, reducing the number of elements in the run by 1. If it's not, we decrease J by 1 to move on to the previous compressed element. As you can see, the conditions listed above are still true.

Once the loop ends, the first condition of the ones above ensures that all elements have been correctly decompressed.

Note that we never store anything to L_1 itself, only to its elements. This is done to avoid using any more memory than necessary: if we stored to L_1 , a copy of the list would get temporarily stored to Ans, and we would be using twice the memory we need. This way, the routine will work even for large lists. As a bonus, the only time we change the size of the list is the very beginning. So if the decompressed list wouldn't fit in memory, the routine crashes immediately and keeps the list intact.

Error Conditions

- **ERR:MEMORY** is thrown if there is not enough space to store the decompressed list.
- **ERR:INVALID DIM** is thrown if the decompressed list would be longer than 999 elements.

Related Routines

- RLE Compression

For the most up-to-date version of this routine, see <http://tibasicdev.wikidot.com/rle-decompress>

Scramble A String

This routine takes a string stored in Str1 and scramble it. The results is contained in Ans. For example, "ABCDE12345" would be "B34AC1DE25"

```
:length(Str1→L
:cumSum(binomcdf(Ans-1,0→L1
:rnd(L→L2
:SortA(L2,L1
:sub(Str1,L1(1),1
```

```
:For(θ,2,L
:Ans+sub(Str1,L1(θ),1
:End
```

With your routine stored in Str1, it creates L₁: {1,2,3,4,...L}. After that, L₂ is created randomly to sort L₁ in function of L₂. L₁ now looks like {5,3,4,1,2} if you entered a 5 character string. In the For(loop, it takes one by one the character of Str1 accordingly to L₁ to store it to Ans. Your scrambled string is now in Ans.

Now, I give you another code for this routine that apparently works as well. This one is 5 bytes smaller. I did test and the speeds are quite the same.

```
:rand(length(Str0→A
:cumSum(1 or Ans→B
:SortA(LA,LB
:sub(Str0,LB(1),1
:For(N,2,length(Str0
:Ans+sub(Str0,LB(N),1
:End
```

Related Routines

- [Pad a String](#)
- [Reverse a String](#)

For the most up-to-date version of this routine, see <http://tibasicdev.wikidot.com/scramble-a-string>

Shading Circles

```
:For(N,0,R,ΔX
:√(R2-N2
:Line(A+N,B-Ans,A+N,B+Ans
:Line(A-N,B-Ans,A-N,B+Ans
:End
```

Although it is possible to shade in a circle using the Shade(command (i.e., Shade(-√(R²-(X-A)²)+B,√(R²-(X-A)²)+B)), that is actually quite impractical in a real program because of its slow speed. Fortunately, you can improve upon that by using your own routine.

When graphing a circle, there are a few main things you need to know: the radius and the (X,Y) coordinates of the center. In our routine, the R variable is the radius, the A variable is the circle's X

Routine Summary

Scrambles a string

Inputs

Str1 - The string you want to scramble

Outputs

Ans - The scrambled string

Variables Used

Ans, Str1, L, θ, L₁, L₂

Calculator Compatibility

TI-83/84/+/SE

Authors

seb83, Edward H, Timothy Foster

Download

[scramble_prgm.zip](#)

Routine Summary

Shades in a circle.

Inputs

R - the radius of the circle

A - the X coordinate of the circle's center

B - the Y coordinate of the circle's center

Outputs

None

coordinate and the B variable is the circle's Y coordinate.

Rather than displaying the circle as one big circle, we are going to display it line by line using a For(loop, starting from the center. The $\sqrt{(R^2-N^2)}$ formula is based on the circle formula $R^2=(X-H)^2+(Y-K)^2$, with the formula rearranged to get the respective part of the circle.

The circle should display pretty quickly, but it all depends on the values that you chose for the variables; smaller values will be faster, and likewise larger values will be slower. In addition, the circle may not display correctly if you don't have the right graph settings, so you should set your calculator to a friendly graphing window.

For the most up-to-date version of this routine, see <http://tibasicdev.wikidot.com/shading-circles>

String to List

```
:DelVar L1
:For(A,1,length(Str1
:inString("ABCDEFGHIJKLMNPQRSTUVWXYZ
:End
```

With our characters stored in Str1, we loop through each character and store its position in our reference string (the uppercase alphabet) to the respective element of L₁. In order to construct a list with all of the positions of the characters, we first need to create a new list. This is what the DelVar L₁ is used for.

While using DelVar to create a list sounds like an oxymoron, the reason that it works is because the calculator automatically creates the next element in a list when you try to access it. For example, the first time through the For(loop, we tell the calculator to store the position of our first character to the first element in L₁. Since there are no elements in L₁, the calculator creates a new element at the beginning of the list, and assigns it the value.

This routine only allows for values from 1 to 26 in the list, since our string of characters is the uppercase alphabet, and each list value must match up to one of the string positions. If you add more characters to the string, however, you can increase the range of values in the list. This routine uses Str1, so you should clean it up at the end of your program.

Variables Used

N, R, A, B

Calculator Compatibility

TI-83/84/+/SE

Author

Jutt

URL: [United TI](#)

Download

[shadecircles.zip](#)

Routine Summary

Converts a string to a list of numbers.

Inputs

Str1 - The string you want to convert

Outputs

L₁ - The list that the numbers are stored to

Variables Used

L₁, A, Str1

Calculator Compatibility

TI-83/84/+/SE

Download

[stringtolist.zip](#)

Related Routines

- [Number to String](#)
- [List to String](#)

For the most up-to-date version of this routine, see <http://tibasicdev.wikidot.com/string-to-list>

Strip a String

```
:While " "=sub(Ans,1,1) or " "=sub(A  
:sub(Ans,1+( " "=sub(Ans,1,1)),length  
:End
```

With our string stored in Ans, we check to see if there is a space at the beginning or end of the string. If there is a space, then we remove it by storing the substring of the string that doesn't include the first character (for a space at the beginning) or the last character (for a space at the ending), and store the new string to Str1.

At the same time, the new string is stored to Ans, which is what we use next time through the While loop. The loop gets repeated over and over again until we have stripped all of the spaces from the beginning and end of the string. If you only want to remove spaces on one side of the string, edit the first and second lines accordingly.

Using Ans allows us to not have to use another string variable, since Ans can act like a string and it gets updated accordingly, and Ans is also faster than a string variable. In addition, by storing the string to Str1 inside the loop, we avoid having to add an additional storage to the routine.

We store the string to a more permanent variable (in this case, Str1) for future use. When you are done using Str1, you should clean it up at the end of your program.

Related Routines

- [Pad a String](#)
- [Scramble a String](#)

For the most up-to-date version of this routine, see <http://tibasicdev.wikidot.com/strip-a-string>

Sum of Digits

Routine Summary

Strip a string of its spaces on the left and right.

Inputs

Ans - The string you want to strip

Outputs

Str1 - The stripped string

Variables Used

Ans, Str1

Calculator Compatibility

TI-83/84/+/SE

Download

[stripstring.zip](#)

```
:sum(int(10fPart(Xseq(10^(I-1), I,-ir
```

With our number stored in X, we loop through each of the digits of the number starting from the right using the seq(command. We get the respective digit by raising the number to the respective power of 10 and then multiplying the result by 10. The digits are returned separately in a list, and then we take the sum of them using the sum(command.

For example, if the number is 1234, we get a list of {1,2,3,4}, which then returns a sum of 10. You should note, though, that this routine only works with positive and negative whole numbers, and will return incorrect results if you try to use a decimal number. You could fix this problem by using the code below on the input:

```
:abs(E13(X/10^(int(log(abs(X+not(X→X
```

Here, we obtain the appropriate power of ten by which to divide the number so as to leave only one digit to the left of the decimal point. Because a real variable may contain only 14 digits, the answer is multiplied by 10^{13} to guarantee the removal of any fractional part. From taking the absolute value of this new result, our number can be safely entered into the routine above.

Note: It should be understood that this routine is only capable of adding the first 14 digits of a number. An input of π , for instance, would return 69, but the entire digital sum would actually be considerably larger than that (∞).

Related Routines

- [Number Concatenation](#)
- [Number Subset](#)

For the most up-to-date version of this routine, see <http://tibasicdev.wikidot.com/sum-of-digits>

Sum of Matrix Elements

```
:dim([A]
:Matr►list(cumSum([A])^T,Ans(1),L1
:sum(L1
```

The cumSum(command gets the cumulative sum of each column in the matrix, adding the value of the previous element to the next element, and repeating this for each consecutive element in the column.

Routine Summary

Returns the sum of digits of a number.

Inputs

X - the number you want

Outputs

Ans - the sum of X's digits

Variables Used

X, I, Ans

Calculator Compatibility

TI-83/84/+SE

Author

DarkerLine

URL: [United TI](#)

Routine Summary

Returns the sum of the elements of a matrix.

Inputs

[A] - the matrix whose elements you want to sum

When the `cumSum(` command is finished, the last element in each column will contain the sum of that column. Taking the `T` (transpose) of the resulting matrix switches columns and rows.

Then, using the `Matr►list(` command, the last column of this result is stored to `L1`. This column was originally the last row of `cumSum(`'s output, so it contains all the column sums. Finally, by calculating the sum of that list, we get the total of the matrix elements. `L1` is no longer necessary, and can be deleted.

If you want to use the sum for future use, you can store it to a more permanent variable, such as `A` or `X`. When you are done using `[A]`, you should clean it up at the end of your program.

want to sum

Outputs

`Ans` - the sum of the matrix elements

Variables Used

`[A], L1, Ans`

Calculator Compatibility

TI-83/84/+/SE

Author

zada

URL: [United TI](#)

Download

[sumofmatrix.zip](#)

For the most up-to-date version of this routine, see <http://tibasicdev.wikidot.com/sum-of-matrix>

Typewriter Routine

```
:For(T,1,length(Str1  
:Text(A,B,sub(Str1,1,T  
:rand(5  
:End
```

We use a `For(` loop over the length of the string to go through the string letter by letter. In order to make the spacing come out right, instead of figuring out the right coordinate to display each character at, we display the first `T` characters of the string starting from the beginning. The `rand(5)` provides a split-second delay to achieve the typewriter effect.

For multiple lines with the typewriter effect, you can combine this routine with the one to wordwrap text.

Error Conditions

- ERR:DOMAIN is thrown if the string doesn't fit entirely on the screen.

Related Routines

- Wordwrapping Text

Routine Summary

Makes text appear letter by letter

Inputs

`Str1` - Text to be displayed.
`A, B` - Row and column to display text.

Variables Used

`A, B, T, Str1`

Calculator Compatibility

All TI calculators

Wordwrapping Text on the Graphscreen

```
:1→C  
:For(A,A,57,6  
:inString(Str1,"/",Ans  
:Text(A,B,sub(Str1,C,Ans-C  
:If Ans=length(Str1:57→A  
:Ans+1→C  
:End
```

This routine displays a string on the graph screen, with line breaks after every slash "/" character. The routine requires there to be a slash at the end of the string (so, a sample string would be "THIS TEXT IS TOO LONG TO/DISPLAY ON ONE LINE/").

The row and column (A and B) inputs determine the upper and left boundaries of the text. You determine the right boundary by where you put the slash characters. The lower boundary is 57, hardcoded into the routine (this is the lowest that text can be displayed), but you could change this if you wanted a different boundary.

The variable C is used for an index inside Str1, that tells us which character we're currently on. We're starting at the beginning of the string, so we store 1 to C.

The For(loop will increment A by 6 each time we display a line (small font characters are 6 pixels tall). We start at the current value of A, and end at 57 because beyond that, Text(will throw a domain error.

Now, we search for the next slash "/" character using the inString(command. Then, using Text(, we display the line between C and the "/", not including the slash itself. Ans+1→C will move the C index to the next character after the slash.

The line If Ans=length(Str1:57→A will end the loop early as soon as we reach the end of the string. This command can modify Ans, which would be bad, but it only does so when we're ready to exit anyway — at that point, we won't be needing Ans anymore.

Error Conditions

- **ERR:INVALID DIM** is thrown if Str1 is improperly formatted (check for the backslash at the end).
- **ERR:DOMAIN** is thrown if text goes off the screen. This shouldn't happen unless B is very close to the right edge, if A or B are negative, or if the calculator is in Horiz mode (in

Routine Summary

Displays text on several lines on the graph screen.

Inputs

Str1 - text to display
A - starting row
B - starting column

Outputs

None

Variables Used

Str1, *A*, *B*, *C*, *Ans*

Calculator Compatibility

TI-83/84/+/SE

Download

[wordwrap.zip](#)

which case, replace 57 with 25).

Related Routines

- [Highlighting Text](#)

For the most up-to-date version of this routine, see <http://tibasicdev.wikidot.com/wordwrap-text>

Abbreviations

#|A|C|D|E|F|G|I|L|M|O|P|R|S|T|V|Y|Z

#

1337

1337 (pronounced as leet) is primarily an Internet fad of writing words by replacing the standard letters with their alphanumeric counterparts. Within the TI community, this phrase is used to indicate that somebody is an advanced programmer and very knowledgeable.

68K

Motorola 68000 is the microprocessor that the TI-89, TI-92, and Voyage 200 graphing calculators use. Zilog makes the microprocessor for the TI-83 series of graphing calculators.

A

Ans

The last answer that was stored in the calculator. The calculator also keeps a history of the last few statements entered in on the home screen where general usage of the calculator occurs.

API

An Application Programming Interface is a set of methods or calls that the calculators provides to support requests made by programs.

App

A Flash application is an enhanced assembly program, with the primary differences being that it runs directly from the archive and its size is always in 16KB chunks (because of the calculator's internal memory design).

Arc

Archive is the permanent memory (also known as ROM) on the TI calculator. If your calculator crashes, all of your files in RAM will be lost, while everything in the archive will still be intact.

Arg

An argument is an expression that is supplied as input to a program, function, or command.

Asm

Assembly language is the other programming language available on the TI graphing calculators. It is much faster than TI-Basic, but it requires you to use a computer to compile it, which can be hazardous (namely, if you screw up in assembly, you can cause real damage to your calculator).

C

Cmd

A command is an instruction telling the calculator to do a particular task.

Cond

A condition is a statement or expression that evaluates to a true (1) or false (0) value.

D

Dec

A decimal is a number that has one or more digits after the period (i.e., 1.2345).

E

EOS

The Equation Operating System (EOSTM) is the routine in the calculator that determines order of operations.

Equ

An equation is a mathematical expression involving variables and/or functions.

Err

An error is when an unexpected condition occurs, and the calculator displays a message to attempt to tell you what went wrong. Errors run the gamut from serious to stupid, meaning it might be time to buy a new calculator or you might have simply misused a command.

Expr

A combination of numbers, variables, and functions.

F

Frac

A fraction is the result of division of two numbers (i.e., 5/7 or 1/3), including integers or even other fractions.

Func

A function is a command that returns a value of some kind (number, list, string, etc.).

FX

Special graphical or sound effects used in a game.

G

GUI

A graphical user interface is a type of user interface which allows people to interact with the calculator.

I

IDE

An integrated development environment is a software application that provides utilities for program development, usually including an editor and debugger.

Img

An image is a visual depiction of something.

Int

An integer is a whole number, either positive, negative, or zero.

I/O

Input is what you provide to the calculator when you press a key or select a menu, and output is how the calculator responds. The two are interdependent of each other.

L

Lib

A library is a program (usually made in assembly) that is designed to be an auxiliary utility for another program. Libraries many times provide functions that are not supported or not practical in TI-Basic (such as inverting the screen).

M

Matr

A matrix is a two-dimensional list (row by column) used for holding lots of information.

O

Op

An operator is a function that performs a set task.

OS

The operating system is the logic that controls the calculator.

P

Prgm

A program is simply the list of instructions that tells the calculator what to do to perform a task.

Pt

A point is a specific 1×1 space on the calculator's graph screen. You can use the [Pt commands](#) to manipulate a point. The primary difference between a pixel and a point is that a pixel is not affected by the graph screen settings, while a point is.

Pxl

A pixel is a specific 1×1 space on the calculator's graph screen. You can use the [Pxl commands](#) to manipulate a pixel.

R

Rand

A command used to generate a uniformly-distributed pseudo-random number between 0 and 1.

S

SE

Silver Edition is the more powerful version of the TI-83+ and TI-84+ calculators, with more memory and a faster processor.

Str

A string is a variable type that stores text as a series of symbols or tokens.

T

TI

Texas Instruments is the maker of the graphing calculators. In addition to calculators,

they also make an assortment of other computer related products.

TI-83/84+/SE

Shorthand way to refer to the five graphing calculators in the TI-83 series: TI-83, TI-83+, TI-83+SE, TI-84+, and TI-84+SE.

TIOS

The TI operating system is the built-in operating system that controls the calculator.

V

Var

A variable is a reference to the information that it holds. There are several different variables available in TI-Basic, including reals, lists, matrices, and strings.

Y

Y=

The built-in editor which is used to enter and edit the Y# functions.

Z

Z80

Zilog Z80 is the microprocessor that the TI-83 series of graphing calculators use. Motorola makes the microprocessor for the TI-89, TI-92, and Voyage 200 graphing calculators.

For the most up-to-date version of this page, see <http://tibasicdev.wikidot.com/abbreviations>

Binary, Hexadecimal and Octal number system

What?

The binary number system is a number system only using 2 values: 0 and 1 (Lat. “bis” means “twice”). This system is ideal for computer science, 0 representing “no current” and 1 representing “current”.

The hexadecimal number system consists of 16 values: 0,1,2,3,4,5,6,7,8,9,A,b,C,d,E and F (Greek “hexa kai deka” means 16), respectively 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14 and 15. Note that the “b” and “d” are written in lowercase, to make it easier to display on a digital display.

The octal number system uses 8 values (0,1,2,3,4,5,6,7). The name is -again- derived from Greek/Latin: okto(GR)/octo(lat.).

fold

Table of Contents

How?

Conversion.

From decimal to binary

[From binary to decimal](#)
[From decimal to hexadecimal.](#)
[From hexadecimal to decimal](#)
[From decimal to octal](#)
[From octal to decimal](#)
[Fun Facts](#)
[End](#)

How?

Now you might be asking yourself how to *read* these numbers. Well, that's not so difficult. First, I'll give a general mathematical explanation which can be fit into one formula:

$$V = vB^P \quad (1)$$

In human language: the value of the cipher in the number is equal to the value of the cipher on its own multiplied by the base of the number system to the power of the position of the cipher from left to right in the number, starting at 0. Read that a few times and try to understand it.

Thus, the value of a digit in binary **doubles** every time we move to the left. (see table below)

From this follows that every hexadecimal cipher can be split up into 4 binary digits. In computer language: a nibble. Now take a look at the following table:

Binary Numbers				Hexadecimal Value	Decimal Value
8	4	2	1		
0	0	0	0	0	0
0	0	0	1	1	1
0	0	1	0	2	2
0	0	1	1	3	3
0	1	0	0	4	4
0	1	0	1	5	5
0	1	1	0	6	6
0	1	1	1	7	7
1	0	0	0	8	8
1	0	0	1	9	9
1	0	1	0	A	10
1	0	1	1	B	11
1	1	0	0	C	12
1	1	0	1	D	13
1	1	1	0	E	14
1	1	1	1	F	15

Another interesting point: look at the value in the column top. Then look at the values. You see what I mean? Yeah, you're right! The bits switch on and off following their value. The value of the first digit (starting from the right), goes like this: 0,1,0,1,0,1,0,1,... Second digit:

0,0,1,1,0,0,1,1,0,0,1,1,0,0... Third digit (value=4): 0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,... And so on...

Now, what about greater numbers? Therefore we'll need an extra digit. (but I think you figured that out by yourself). For the values starting from 16, our table looks like this:

Binary Numbers						Hexadecimal Value	Decimal Value
16	8	4	2	1			
1	0	0	0	0		10	16
1	0	0	0	1		11	17
1	0	0	1	0		12	18
1	0	0	1	1		13	19
1	0	1	0	0		14	20
1	0	1	0	1		15	21
1	0	1	1	0		16	22
1	0	1	1	1		17	23
1	1	0	0	0		18	24
1	1	0	0	1		19	25
1	1	0	1	0		1A	26
1	1	0	1	1		1B	27
1	1	1	0	0		1C	28
1	1	1	0	1		1D	29
1	1	1	1	0		1E	30
1	1	1	1	1		1F	31

For octals, this is similar, the only difference is that we need only 3 digits to express the values 1->7. Our table looks like this:

Binary Numbers				Octal Value	Decimal Value
4	2	1			
0	0	0		0	0
0	0	1		1	1
0	1	0		2	2
0	1	1		3	3
1	0	0		4	4
1	0	1		5	5
1	1	0		6	6
1	1	1		7	7

Conversion.

In the latter topic I explained the logic behind the binary, hexadecimal and octal number systems. Now I'll explain something more practical. If you fully understood the previous thing you can skip this topic.

From decimal to binary

- Step 1: Check if your number is odd or even.
- Step 2: If it's even, write 0 (proceeding backwards, adding binary digits to the left of the result).
- Step 3: Otherwise, if it's odd, write 1 (in the same way).
- Step 4: Divide your number by 2 (dropping any fraction) and go back to step 1. Repeat until your original number is 0.

An example:

Convert 68 to binary:

- 68 is even, so we write 0.
- Dividing 68 by 2, we get 34.
- 34 is also even, so we write 0 (result so far - 00)
- Dividing 34 by 2, we get 17.
- 17 is odd, so we write 1 (result so far - 100 - remember to add it on the left)
- Dividing 17 by 2, we get 8.5, or just 8.
- 8 is even, so we write 0 (result so far - 0100)
- Dividing 8 by 2, we get 4.
- 4 is even, so we write 0 (result so far - 00100)
- Dividing 4 by 2, we get 2.
- 2 is even, so we write 0 (result so far - 000100)
- Dividing 2 by 2, we get 1.
- 1 is odd, so we write 1 (result so far - 1000100)
- Dividing by 2, we get 0.5 or just 0, so we're done.
- Final result: 1000100

From binary to decimal

- Write the values in a table as shown before. (or do so mentally)
- Add the value in the column header to your number, if the digit is turned on (1).
- Skip it if the value in the column header is turned off (0).
- Move on to the next digit until you've done them all.

An example:

Convert 101100 to decimal:

- Highest digit value: 32. Current number: 32
- Skip the "16" digit, its value is 0. Current number: 32
- Add 8. Current number: 40
- Add 4. Current number: 44
- Skip the "2" and "1" digits, because their value is 0.
- Final answer: 44

From decimal to hexadecimal.

THIS IS ONLY ONE OF THE MANY WAYS!

- Convert your decimal number to binary
- Split up in nibbles of 4, starting at the end
- Look at the first table on this page and write the right number in place of the nibble

(you can add zeroes at the beginning if the number of bits is not divisible by 4, because, just as in decimal, these don't matter)

An example:

Convert 39 to hexadecimal:

- First, we convert to binary (see above). Result: 100111
- Next, we split it up into nibbles: 0010/0111 (Note: I added two zeroes to clarify the fact that these are nibbles)
- After that, we convert the nibbles separately.
- Final result: 27

From hexadecimal to decimal

*Check the formula in the first paragraph and use it on the ciphers in your hexadecimal number.
(this actually works for any conversion to decimal notation)

An example:

Convert 1AB to decimal:

- Value of B = $16^0 \times 11$. This gives 11, obviously
- Value of A = $16^1 \times 10$. This gives 160. Our current result is 171.
- Value of 1 = $16^2 \times 1$. This gives 256.
- Final result: 427

From decimal to octal

- Convert to binary.
- Split up in parts of 3 digits, starting on the right.
- Convert each part to an octal value from 1 to 7

Example: Convert 25 to octal

- First, we convert to binary. Result: 11001
- Next, we split up: 011/001
- Conversion to octal: 31

From octal to decimal

Again, apply the formula from above

Example: convert 42 to decimal

- Value of 2 = $8^0 \times 2 = 2$
- Value of 4 = $8^1 \times 4 = 32$
- Result: 34

Fun Facts

OK, these may not be 100% "fun", but nonetheless are interesting.

- Do you tend to see numbers beginning with 0x? This is common notation to specify hexadecimal numbers, so you may see something like:

0x000000
0x000002
0x000004

This notation is most commonly used to list computer addresses, which are a whole different story.

- This is pretty obvious, but you can "spell" words using hexadecimal numbers. For example:
 - CAB = 3243 in decimal notation.

End

Did you understand everything? If you think so, test yourself:

Bin	Dec	Hex
...	...	3A
...	76	...
101110
...	88	...
1011110
...	...	47

Make some exercises yourself, if you want some more.

For the most up-to-date version of this page, see <http://tibasicdev.wikidot.com/binandhex>

Error Conditions

A | B | D | E | I | L | M | N | O | R | S | T | U | V | W | Z

In the Error Conditions section on a command page, descriptions will be given of the errors that can result when using a command. These do not include errors that can occur with virtually any command, such as ERR:ARGUMENT or ERR:SYNTAX. However, if one of these errors is triggered in an unusual way, it will be listed.

Be aware that certain errors don't occur when graphing or using one of the commands DrawF, DrawInv, Tangent(, or Shade(. Instead, the graph point is skipped and treated as an undefined value. Tangent(is a minor exception, actually: if there's an error at the tangent point, it will be treated normally. The errors that are ignored in graphs are:

- ERR:DATA TYPE
- ERR:DIVIDE BY 0
- ERR:DOMAIN
- ERR:INCREMENT
- ERR:NONREAL ANS
- ERR:OVERFLOW
- ERR:SINGULAR MAT

There are also several error messages that don't actually occur (as far as anyone knows) from normal use of the calculator — however, the messages are stored in the OS

along with the normal messages. It's conceivable that an assembly program, especially an official TI application, would use these error messages for its own purposes. These messages are ERR:SOLVER, ERR:SCALE, ERR:NO MODE, ERR:LENGTH, and ERR:APPLICATION.

A

ARCHIVED

- You have attempted to use, edit, or delete an archived variable. For example, `dim(L1)` is an error if L1 is archived.
- Use UnArchive *variable name* to unarchive the variable before using it. For lists, SetUpEditor is recommended instead since it does the same thing, but works on the TI-83 (which has no archive) as well, and does not crash when the list is undefined.
- There is no way to archive or unarchive programs using pure Basic, although several assembly utilities are available for doing so.

ARCHIVE FULL

- You have attempted to archive a variable and there is not enough space in archive to receive it.

ARGUMENT

- A function or instruction does not have the correct number of arguments. See the appropriate command page.
- A function or instruction that can have any number of arguments has 256 or more.
- In general, if a function has more than one non-SYNTAX error in it, this is the error that will be generated first (if it applies, of course).

B

BAD ADDRESS

- You have attempted to send or receive an application and an error (e.g., electrical interference) has occurred in the transmission.

BAD GUESS

- With the solve(function, the equation solver, or an operation from the CALC menu, your guess wasn't within the lower and upper bound, or else the function is undefined at that point. Change the guess.

BOUND

- In a CALC operation or with Select(, you defined Left Bound > Right Bound.
- In fMin(, fMax(, solve(, or the equation solver, the lower bound must be less than the upper bound.

BREAK

- You pressed the [ON] key to break execution of a program, to halt a DRAW instruction, or to stop evaluation of an expression.

D

DATA TYPE

- You entered a value or variable that is the wrong data type.
- For a function (including implied multiplication) or an instruction, you entered an argument that is an invalid data type, such as a complex number where a real number is required.
- In an editor, you entered a type that is not allowed, such as a matrix entered as an element in the stat list editor.
- You attempted to store an incorrect data type, such as a matrix to a list.
- This error is not returned when graphing (see the note at the top of the page).

DATE

- Only on the TI-84+ or TI-84+ SE, this error occurs when an invalid date is entered.
- Below the error menu, an explanation of what's wrong is given: e.g., "Invalid day for month selected."

DIM MISMATCH

- You attempted to perform an operation that references more than one list or matrix, but the dimensions do not match.
- In most cases, the dimensions are required to be equal, with the following exceptions:
 - When multiplying two matrices, the number of columns of the first must match the number of rows of the second.
 - When using augment(to combine two matrices, only the number of rows must match.
 - With the List►matr(command, the lists don't have to match sizes - shorter lists will be padded with zeroes.

DIVIDE BY 0

- You attempted to divide by zero.
- You attempted a linear regression on data that fit a vertical line.
- This error is not returned when graphing (see the note at the top of the page).

DOMAIN

- You specified an argument to a function or instruction outside the valid range. This error is not returned during graphing. The TI-83+ allows for undefined values on a graph.
- You attempted a logarithmic or power regression with a negative X or an exponential or power regression with a negative Y.
- You attempted to compute Σ Prn(or Σ Int(with pmt2 < pmt1.
- You've assigned a value to n (the sequential graph variable), nMin or nMax that isn't an integer, or that is less than 0.
- This error is not returned when graphing (see the note at the top of the page).

DUPLICATE

- You attempted to create a duplicate group name.
- You attempted to create a duplicate program using AsmComp(.)

Duplicate Name

- A variable you attempted to transmit cannot be transmitted because a variable with that name already exists in the receiving unit.
- Also appears when you unpack a group and a variable in the group is already defined. In both cases, it will give you several options for correcting the error, including Omit, Rename, and Overwrite.

E

EXPIRED

- You have attempted to run a Flash application with a limited trial period which has expired.

Error in Xmit

- The TI-83+ was unable to transmit an item. Check to see that the cable is firmly connected to both units and that the receiving unit is in receive mode.
- You pressed [ON] to break during transmission.
- You attempted to perform a backup from a TI-82 to a TI-83+.
- You attempted to transfer data (other than L1 through L6) from a TI-83+ to a TI-82.
- You attempted to transfer L1 through L6 from a TI-83+ to a TI-82 without using 5:Lists to TI-82 on the LINK SEND menu.

I

ID NOT FOUND

- This error occurs when the SendID command is executed, but the proper calculator ID cannot be found.

ILLEGAL NEST

- You attempted to use an invalid function in an argument to a function.
- This happens when using seq(in the expression for seq(, or expr(inside the string argument of expr(.

INCREMENT

- The increment in seq(is 0 or has the wrong sign.
- The increment in a For(loop is 0.
- This error is not returned when graphing (see the note at the top of the page).

INVALID

- You attempted to reference a variable or use a function where it is not valid. For example, Y_n cannot reference Y , X_{\min} , ΔX , or $TblStart$.
- You attempted to reference a variable or function that was transferred from the TI-82 and is not valid for the TI-83+. For example, you may have transferred $UnN1$ to the TI-83+ from the TI-82 and then tried to reference it.
- In Seq mode, you attempted to graph a phase plot ([uvAxes](#), [uwAxes](#), or [vwAxes](#)) without defining both equations of the phase plot.
- In Seq mode, you attempted to graph a recursive sequence without having input the correct number of initial conditions.
- In Seq mode, you attempted to reference terms other than $(n-1)$ or $(n-2)$.
- You attempted to designate a graph style that is invalid within the current graph mode.
- You attempted to use [Select\(](#) without having selected (turned on) at least one [xyLine](#) or scatter plot.

INVALID DIM

- You tried to access an element past the end of a list or matrix (there is an exception: it's allowed to store to the element one past the end of a list, adding the element).
- You specified dimensions for an argument that are not appropriate for the operation.
- You specified a [list](#) dimension as something other than an integer between 1 and 999.
- You specified a [matrix](#) dimension as something other than an integer between 1 and 99.
- You attempted to invert a matrix that is not square.

ITERATIONS

- The [solve\(](#) function or the equation solver has exceeded the maximum number of permitted iterations. Examine a graph of the function. If the equation has a solution, change the bounds, or the initial guess, or both.
- [irr\(](#) has exceeded the maximum number of permitted iterations.
- When computing $I\%$, the maximum number of iterations was exceeded.

L

LABEL

- The label in the [Goto](#) instruction is not defined with a [Lbl](#) instruction in the program.
- When this error occurs, the option 2:Goto doesn't show up.

M

MEMORY

- Memory is insufficient to perform the instruction or function. You must delete items from memory before executing the instruction or function.
- You might also want to try archiving some variables if you have nothing in RAM that you don't need.
- Recursive problems return this error; for example, graphing the equation $Y_1=Y_1$.
- Branching out of an [If:Then](#), [For\(](#), [While](#), or [Repeat](#) loop with a [Goto](#) will waste memory until the program finishes running, because the [End](#) statement that terminates the loop is never reached. Unless a program is very large, this is one of

the likeliest causes of ERR:MEMORY. Refer to [memory-leaks](#).

MemoryFull

- You are unable to transmit an item because the receiving unit's available memory is insufficient. You may skip the item or exit receive mode.
- During a memory backup, the receiving unit's available memory is insufficient to receive all items in the sending unit's memory. A message indicates the number of bytes the sending unit must delete to do the memory backup. Delete items and try again.

MODE

- You attempted to store to a [window variable](#) in another graphing mode or to perform an instruction while in the wrong mode; for example, [DrawInv](#) in a graphing mode other than [Func](#).

N

NO SIGN CHNG

- The [solve](#)(function or the equation solver did not detect a sign change.
- You attempted to compute I% when FV, (N*PMT), and PV all share the same sign.
- You attempted to compute [irr](#)(when CFList and CFO share the same sign.

NONREAL ANS

- In [Real](#) mode, the result of a calculation yielded a complex result.
- This error is not returned when graphing (see the [note](#) at the top of the page).

O

OVERFLOW

- You attempted to enter, or you have calculated, a number that is beyond the range of the calculator (-1_E100 to 1_E100, non-inclusive).
- Sometimes you can fix this error by re-ordering operations. For example, 60!*30!/20! will return an overflow error, but 60!/20!*30! will not.
- This error is not returned when graphing (see the [note](#) at the top of the page).

R

RESERVED

- You attempted to use a [system variable](#) inappropriately (for example, performing [1-Var Stats](#) on the reserved list [L RESID](#)).

S

SINGULAR MAT

- A singular matrix (determinant = 0) is not valid as the argument for $^{-1}$.
- A regression generated a singular matrix (determinant = 0) because it could not find a solution, or a solution does not exist.
- This error is not returned when graphing (see the note at the top of the page).

SINGULARITY

- The expression in the solve(function or the equation solver contains a singularity (a point at which the function is not defined). Examine a graph of the function. If the equation has a solution, change the bounds or the initial guess or both.
- Although the correct spelling is "singularity", the error message shown has "singularity" on all calculators and OS versions.

STAT

- You attempted a stat calculation with lists that are not appropriate.
- Statistical analyses must have at least two data points.
- Med-Med must have at least three points in each partition.
- When you use a frequency list, its elements must be at least 0.
- $(X_{\text{max}} - X_{\text{min}}) / X_{\text{scl}}$ must equal 47 for a histogram.

STAT PLOT

- You attempted to display a graph when a stat plot that uses an undefined list is turned on.
- To fix this error, use the command PlotsOff to turn off plots when you're using the graph screen.

SYNTAX

- The command contains a syntax error. Look for misplaced functions, arguments, parentheses, or commas. See the appropriate command page.
- This error will also occur in place of a DATA TYPE error if the variable type in question is a variable name (with seq(, solve(, For(, and other commands)
- The command was attempting to get expr(of a non-value string, i.e., trying to evaluate a space, equals sign, etc.

T

TOL NOT MET

- You requested a tolerance to which the algorithm cannot return an accurate result.

U

UNDEFINED

- You referenced a variable that is not currently defined. This error doesn't occur with number variables (A-Z,θ), which have a default value of 0.
- Lists, matrices and strings have to be stored to first, in order to be used.
- Most system variables always have a value, so this error doesn't apply to them.

- However, statistical variables are undefined except for those used by the last relevant command.
- Undefined equation variables return ERR:INVALID instead.

V

VALIDATION

- Electrical interference caused a link to fail or this calculator is not authorized to run the application.

VARIABLE

- You have tried to archive a variable that cannot be archived or you have tried to unarchive an application or group.
- Variables that cannot be archived include:
 - The number variables R, T, X, Y, and θ (because they are used for graphing)
 - The list LRESID (because it's reserved for residuals from regression models)
 - System variables (including statistical variables, finance variables, equations, plots, and window variables)
 - The AppIdList.

VERSION

- You have attempted to receive an incompatible variable version from another calculator.

W

WINDOW RANGE

- A problem exists with the window variables.
 - You defined $X_{\min} > X_{\max}$ or $Y_{\min} > Y_{\max}$.
 - X_{\min} and X_{\max} (or Y_{\min} and Y_{\max}) are equal or so close that numerical precision can't allow for enough values between them.
 - The values for θ_{\min} , θ_{\max} , and θ_{step} create an empty or never-ending loop for θ .
 - The values for T_{\min} , T_{\max} , and T_{step} create an empty or never-ending loop for T .

Z

ZOOM

- A point or a line, instead of a box, is defined in ZBox.
- A ZOOM operation returned a math error.

For the most up-to-date version of this page, see <http://tibasicdev.wikidot.com/errors>

Key Codes

The picture to the right top contains the value returned by `getKey` for each keypress. The picture is organized so that the key codes are placed where they would be on the literal calculator. You should note that the ON key (the key in the bottom left corner) has no key code, so you cannot check for nor disable it.

If you look at the key codes, you will probably notice that they actually follow a pattern: a key is represented by putting its row and column together. For example, the ENTER key is row 10 column 5, so its value is 105. The arrow keys look like they would be numbered separately from the other keys, but they actually follow the same pattern.

In case you want to know the key codes while using your calculator, here is a simple program to use:

```
:Repeat Ans=105  
:getKey  
:If Ans:Disp Ans  
:End
```

The picture to the right bottom contains the value returned by the `real(8` command when using `xlib`. The key placement is somewhat disorganized, but consistent for the most part. The ON key will return a value, according to the tutorial, but due to the ON break inherent to TI-Basic, it will not be returned.

11	12	13	14	15
21	22	23	24	25
31	32	33		34
41	42	43	44	45
51	52	53	54	55
61	62	63	64	65
71	72	73	74	75
81	82	83	84	85
91	92	93	94	95
	102	103	104	105

53	52	51	50	49
54	55	56	2	4
48	40	32		3
47	39	31	23	15
46	38	30	22	14
45	37	29	21	13
44	36	28	20	12
43	35	27	19	11
42	34	26	18	10
41	33	25	17	9

For the most up-to-date version of this page, see <http://tibasicdev.wikidot.com/key-codes>

Tokens and Token Size

Each command, variable, and operation on the TI-83 series calculators is represented by a "token." This means that internally, the calculator does not store a command such as "cos(" as the letters c, o, s, and (. It stores a single number that it will later translate as "cos(" when necessary. In this case, the value is 196, but you most likely don't need to know that.

What you do need to know is that not all tokens are the same size. If there were 256 tokens or less, then you could fit all their values into 1 byte and be happy. Unfortunately, the TI-83 has more than 256 commands and variables. Therefore TI employed some trickery and made some tokens take up 1 byte (usually the most common ones, though they seem to have had a

different idea of "common") and some take up 2 bytes.

What this means to you, as the programmer, is that the size of the program is determined by the number of commands, not the number of letters in it: a short line can take up more memory than a longer one if it uses a lot of commands. Furthermore, some commands will take up the memory of two commands rather than one, so a line with a few of these commands may take up as much memory as a line with more commands of the ordinary type.

Lowercase letters are the epitome of memory wasters: at a single character, they each take up 2 bytes of memory. A program that uses a lot of lowercase letters can fill up all of RAM very quickly! This may be avoided by using uppercase letters instead, which only take up 1 byte each. You can also save memory by replacing words such as "If", "or", "and" with the appropriate commands, when displaying text. Such a command will only take up 1 byte, whereas the text may be much larger memory-wise.

Token Tables

These token tables are unnecessary for TI-Basic programmers; they would most likely be of use for someone writing a TI-Basic program editor.

- [One-byte tokens](#)
- [Two-byte user variable tokens](#)
- [Two-byte statistical variable tokens](#)
- [Two-byte window and finance variable tokens](#)
- [Two-byte graph format tokens](#)
- [Miscellaneous two-byte tokens](#)
- [TI-84+ only two-byte tokens](#)

There are several ways to insert a token given its hex value; all of them require an assembly program of some form. One of the easiest is to create an assembly program using `AsmPrgm` followed by the hex codes you want to convert to tokens, assemble it using `AsmComp()`, and then unlock it (there are many programs that allow you to do this).

For the most up-to-date version of this page, see <http://tibasicdev.wikidot.com/tokens>