What Is TI-BASIC?

The primary programming language for the TI calculators because it is easy to learn and use. Although it is capable of advanced games, it is best suited for simple, text-based programs because it is especially slow with rendering graphics and getting user input.

Programs *Name=TESTPROG*

Definition: An organized, step-by-step set of instructions that, when executed, causes the calculator to behave in a predetermined manner.

How to Create a Program:

1. Go to the PRGM menu (press PRGM).
2. Scroll over to NEW and press ENTER.
3. Type in the name of your program.
4. Press ENTER to confirm the program name.

Rules for Naming Programs:

1. Program names can only be up to eight characters long.
2. The characters must be A-Z or θ.
3. The first character must be a letter.
4. Each program must have its own specific name (no duplicates).

Hints:

1. You should choose a program name that actually relates to your program (such as its title).
2. If your program is insignificant (such as a subprogram for a larger program), start it with θ or Z so that it appears at the bottom of the program list.

Data Types *10 7.52 "Hello"*

Integer: Positive or negative numbers with no fractional parts or decimal places. Zero is an integer, too.

Floating Point: Positive or negative numbers that contain a decimal point or exponential notations.

String: A sequence of characters surrounded by quotes.

Boolean: The logical values true/false used to compare data or make decisions.

Operators *+ and =*

Arithmetic

+	Adds two numbers
-	Subtracts one number from another
*	Multiplies two numbers
/	Divides one number by another

Logical

and	Compares two operands; returns true if both are true, otherwise returns false
or	Compares two operands; returns true if either operand is true, otherwise returns false
xor	Compares two operands; returns true if either operand is true, but not both
not	Returns false if its operand is true, otherwise returns true

Comparison

=	Returns true if the operands are equal
≠	Returns true if the operands are not equal
>	Returns true if the first operand is greater than the second
≥	Returns true if the first operand is greater than or equal to the second
<	Returns true if the first operand is less than the second
≤	Returns true if the first operand is less than or equal to the second

Hints:

1. *and* has a higher importance than *or*

Boolean Logic *1 0*

Definition: Based on the principle that a conditional can only be true or false. A true value is represented by 1 or any nonzero number. A false value is represented by 0. Not returns 0 if the number is not zero.

The Truth Table:

A	B	and	or	xor
1	1	1	1	0
1	0	0	1	1
0	1	0	1	1
0	0	0	0	0

Storing & Deleting *DelVar →*

Storing: Values are stored to and recalled from memory using variable names. When an expression containing the name of a variable is evaluated, the value of the variable at that time is used.

Format:

Value→Variable
Expression→Variable

Example:

8→A
3X+2→B

Deleting: The contents of the variable are deleted from memory. The variable is automatically set to zero the next time that it is used. This works for every variable.

Hints:

1. The DelVar command doesn't need a newline/colon following the variable name, so you can take the command on the next line and link it to the end of the variable.

Format:

DelVar variable
DelVar variableDelVar variable

Example:

DelVar X
DelVar AOutput(2,3,"Hello

Menus *Menu*

A menu screen at the beginning of a program is sometimes needed. If you want to organize your program so that you can control program flow into the different parts of it, then use the Menu command. Although it is a generic menu and it only works on the homescreen, it is more economical than using simple text with Goto/Lbl.

If Menu is encountered during the program, the menu screen is displayed with the specified menu title and menu items, the pause indicator turns on, and execution pauses until you select a menu item. The menu title is displayed at the top of the screen and it must be enclosed in quotation marks. You can have up to seven pairs of menu items. Each pair comprises a text item that will be displayed on the screen, and a label item that will be branched to if you press ENTER on it or press its corresponding number.

Hints:

1. Strings will work for the menu title and the menu item text.
2. Variables won't work for the menu item label.
3. Using the Menu command doesn't clear the homescreen; it merely displays the menu screen. This allows you to not have to put ClrHome before it.

Format:

Menu("title","text1",label1,…)

Example:

Menu("Choose","Right",1,"Wrong",2

Pausing *Pause*

If you want to suspend the execution of your program at a certain point, use the Pause command. During the pause, the pause indicator is on in the top-right corner. The user needs to press ENTER to resume execution. The Pause command has an optional argument that can be either a variable or text. The value can be scrolled right/left if it is wider than the screen.

Hints:

1. If you have a Disp statement before the pause, you can take the text/variable from the Disp command and put it as the optional argument for the Pause. This allows you to delete the Disp command.

Format:

Disp "Hello World!
Pause

Example:

Pause "Hello World!

User Input *Input Prompt*

Definition: Getting input from the user is a basic part of any program. It provides a way of determining the direction that the program takes or changing certain variables.

Types: Input, Prompt

Input: Asks the user for the value of some variable and it stores the inputted value for the variable. If the variable is a string, the user must put quotes around the value. The same thing needs to be done for a list ({} instead) and a matrix ([] instead). The Input command allows you to put an optional text display, giving the user a better understanding of what to put for input.

Hints:

1. If you leave off the optional text display it will just put a question mark on the screen.
2. Only the first 16 characters of the optional text display will be shown on the screen.
3. You can store text to a string and use that for the optional text display. If you use the text a lot, it might be smaller.

Format:

Input variable
Input "text",variable
Input Strn,variable

Example:

Input A
Input "Your Name?",Str1
Input Str1,[A]

Prompt: Like Input, it will ask the user for the value of some variable, and it will work with the same variables (with the same rules applying). It allows you to ask for a series of values to be stored in the respective variables.

Hints:

1. Prompt will print the variable that you are asking the user for. This can be a useful alternative to using the optional text display that the Input command has.

Format:

Prompt variableA,variableB,variableC

Example:

Prompt A,Str1,[A]

Displaying Text *Disp Output*

Definition: Displaying text is a fundamental part of almost all programs. Most will have information that needs to be displayed on the screen, either for instruction for the user or for giving the screen a more graphical appearance.

Types: Disp, Output

Disp: Displays the value of each argument after the command on the homescreen. If the value is a variable, the current value will be displayed. If the value is an expression, it is evaluated and the result is displayed on the right side of the next line. If the value is text within quotation marks, it is displayed on the left side of the current line.

Hints:
1. You can link Disp commands together by adding the arguments from the succeeding Disp commands to the first command, separating each one by a comma.
2. The text does not wrap around to the next line if it is more than 16 characters.

Format:
```
Disp valueA,valueB,valueC
```
Example:
```
Disp "Hello",Str1,A
```

Output: Like the Disp command, it will display the value of the argument after the command on the homescreen. It improves upon Disp, though, by allowing you to display text at a specific location. You can specify the row and column. The screen has eight rows (1-8) and sixteen columns (1-16), going from top-down and left-right.

Hints:
1. The Output command can only have one argument; no linking like Disp.
2. Any existing text on the screen will be overwritten unless you precede the Output command with ClrHome.
3. The text will wrap around to the next line if it is more than 16 characters.
4. Any text past row 8, column 16 will not show on the screen.
5. If you put Output(1,1,"") at the end of your program, it will get rid of the "Done" that appears after your program is done.

Format:
```
Output(row,column,value)
```
Example:
```
Output(3,2,"Hello World!
```

Reals *A-Z θ*

Real variables are used for storing numbers. There are 27 letter variables (A-Z and θ) that can be used, as well as the Finance and Window setting variables. Values are stored to and recalled from memory using variable names. When an expression containing the name of a variable is evaluated, the value of the variable at that time is used. You can archive real variables so that they won't be edited or deleted inadvertently.

Hints:
1. Avoid using X and Y for anything other then temporary counters because they change whenever the graphscreen is accessed.
2. Once you have started using variables for certain functions, make sure to continue using them for those functions.

Loops *For While Repeat*

Definition: Loops cause a segment of code to repeat until a stated condition is met. Making sure that the appropriate loops are used is very important.

Types: For, While, Repeat

For: The loop is repeated a specified number of times. It has four arguments: the variable (A-Z or θ), the beginning value, the ending value, and the increment. The increment is optional (the default is 1) and it can be negative. This is the fastest loop.

Format:
```
For (variable,start,end[,increment])
  Command
  Command
End
```
Example:
```
For (X,1,16
 Output(1,X,"X
 Output(8,X,"X
End
```

While: The loop will continue while the conditional is true. The conditional is tested at the beginning of the loop, so if the conditional is false the loop will be skipped entirely. To ensure that the loop will be gone through, you should declare the values of the variables in the conditional before the loop. You should also watch out for infinite loops, which have a stopping condition that will never be reached.

Format:
```
While (condition)
  command
  command
End
```
Example:
```
While X<100
Input "Guess?",X
End
```

Repeat: The loop will continue until the conditional is true. The conditional is tested at the end of the loop, so the loop will be gone through at least once. This allows you to not have to declare the values of variables in the conditional. Like While loops, watch out for infinite loops.

Format:
```
Repeat (condition)
  command
  command
End
```
Example:
```
Repeat Ans
getKey→K
End
```

Exiting Programs *Return Stop*

There are two different commands that you can use for terminating the execution of a program: *Return* and *Stop*. Return stops only the current program and allows any program that called your program to continue running. Stop causes all of the programs to quit and it returns you to the homescreen (unless your program was called from an Assembly program or shell.) So, you should use Return instead of Stop. You don't have to use either of these commands if you can organize your program so that it just naturally quits. If the calculator reaches the end of a program, it will automatically stop executing.

Program Flow *If Then Else End*

Definition: Statements and structures used to change the order in which the program operations will occur. The program can carry out different actions depending on a condition.

Types: If, If-Then-End, If-Then-Else-End

If: A conditional branching statement used to determine whether a stated condition is true. If it is, it will execute the single command on the next line.

Format:
```
If (condition)
  command
```
Example:
```
If A>90
Disp "Good Job!
```

Hints:
1. If you are only changing a variable, you don't need to include the If statement. This works really good for keypresses.

Example:
```
B+2(A=100→B
```

If-Then-End: Like the If branching statement, it will check if the stated condition is true. Unlike If, however, it will execute all of the commands before the End statement, instead of just the command on the next line.

Format:
```
If (condition)
Then
  command
  command
End
```
Example:
```
If K=21
Then
ClrHome
Stop
End
```

If-Then-Else-End: A conditional branching statement that includes a path to follow if the condition is true and a path to follow if the condition is false. All of the commands before the Else will be executed if the condition is true, while all of the conditions after the Else will be executed if it is false.

Format:
```
If (condition)
Then
  command
Else
  command
End
```
Example:
```
If AB
Then
"Hello!→Str1
Else
"Goodbye!→Str1
End
```

Hints:
1. If the commands after the Else consist only of store statements that are different from the store statements before the Else, you don't need the Then-Else-End.

Example:
```
 "Goodbye!→Str1
 If AB
 "Hello!→Str1
```


Lists *dim Fill seq*

Lists are the most versatile variable. A list is just a group of numbers that is contained within one variable. Lists are the most secure variable because you can make your own whereas all the other variables are shared by every program. There are six built-in lists (L1-L6). A custom list can be one to five characters, comprised of A-Z, θ, and numbers, but it must begin with a letter or θ. Lists can be up to 999 elements long. There are several commands that you can use with lists.

Hints:
1. Avoid using lists other than L1-L6.
2. Make your custom list names as short as possible.
3. Use the dim command as an installation flag to check if your program is installed.

Types: L, dim, Fill, augment, seq, sortA, sortD

L: This command is used to identify a user created list. If you create a list using the curly braces and separating each element with a comma, then you don't need the L.

Example:
{1,2,3,4,5→SCORE

dim: Used to find the dimensions of the list, it returns the number of elements of the list. You can use dim with L to create a custom list. When you first create a custom list all of the elements are zeros. You can use dim to redimension an existing list. The elements in the old list that are within the new dimension are not changed. The extra elements are filled by zero. Elements in the old list that are outside the new dimension are deleted.

Example:
3→dim(LHIGH

Fill: Replaces each element in the list with the specified value. This command is preferred over ClrList.

Example:
Fill(5,LNEW

augment: Adds the elements of one list to another. You can also put variables in between curly braces and separate them with a comma and use them as a list. This is useful for saving.

Example:
augment(L1,{A,B,C,D→L2

seq: Returns a list in which each element is the result of the evaluation of an expression with regard to a variable for the values ranging from begin to end at the steps of an optional increment. The default for the increment is one and the increment can be negative. The variable doesn't need to be defined.

Example:
seq(3X,X,1,5→L1

SortA/*SortD*: Sorts the list elements in ascending/descending order. With one list, it sorts the elements of the list and updates it in memory. With two or more lists, it sorts the main list and then sorts all the other lists by placing each element in the same order as the main list. All lists must have the same dimension.

Example:
SortA(L1,L2
SortD(L1,L2

Math *rand abs int*

There are several math functions that programs can use. These functions can be used for making random input in a game or for making math programs. This list doesn't cover all of them, but it does cover the most useful ones.

Types: rand, randInt, iPart, fPart, abs, round, int, min, max

rand: Generates and returns one or more random numbers between zero and one. The optional argument allows you to specify how many random numbers you want. You can multiply rand by another number to get a random number greater than one. To change the initial seed value, store any nonzero number to rand. Just store zero to rand to restore the original seed value. The seed value affects randInt.

randInt: Generates and returns one or more random integers between (and including) specified lower and upper integer bounds. You can generate a list of random numbers by specifying an integer greater than one for the number of trials (the optional third argument).

iPart: Returns the integer part of numbers, expressions, lists and matrices.

fPart: Returns the fractional part of numbers, expressions, lists and matrices.

abs: Returns the absolute value of numbers, expressions, lists and matrices.

round: Returns a number, expression, list or matrix rounded to a user specified number of decimals. If the number of decimals argument is omitted, the value is rounded to the digits that are displayed.

int: Returns the largest integer. For some given values, the result of int is the same as the result of iPart.

min: Returns the smaller of two values or the smallest element in a list. If two lists are compared, min returns a list of the smaller value of each pair of elements. If list and value are compared, min compares each element in list with value.

max: Returns the larger of two values or the largest element in a list. If two lists are compared, max returns a list of the larger value of each pair of elements. If list and value are compared, max compares each element in list with value.

Clearing The Screen *ClrHome*

It is often useful to clear the screen before running a program, during the execution of a program, or after a program finishes running. The command to do this is *ClrHome*. One of the best places to put a ClrHome command is right before displaying text/variables. This ensures that there is nothing else on the screen.

Strings *Str1-Str0*

Strings are used for manipulating text. A string is a sequence of characters enclosed within quotes. Strings can hold as many characters as the amount of free RAM will allow. You can put numbers, letters, and even commands in a string. Each command is counted as one character. The two characters that you can't use are quotes and the store command. There are ten built-in string variables (Str1-Str0). There are several commands that you can use with strings.

Hints:
1. You can combine two or more strings by putting the + operator between them.
2. Use strings and the Output command to make maps on the homescreen.
3. Store commands in a string instead of writing the separate characters of the command.

Types: length, sub, inString, expr

length: Returns the number of characters in a string. The string can be a string of text or a string variable.

Example:
length(Str1

sub: Returns a string that is a subset of an existing string. The string can be a string of text or a string variable. There are three arguments: the string, the start position of the first character of the string, and the number of characters that is wanted.

Example:
sub("HelloWorld",1,5→Str0

inString: Returns the character position in a string of the first character of the substring. The string can be a string of text or a string variable. There are two required arguments and an optional third argument: the string, the substring text, and the character position at which to start the search. The default is one. If the string does not contain the substring or the start is greater than the length of the string, inString returns zero.

Example:
inString("Welcome Back","Back

expr: Converts the character string contained in the string to an expression and executes it. The string can be a string of text or a string variable.

Example:
expr(Str1

Ans *A-Z Str1-Str0 L1-L6 [A]-[J]*

The Ans variable is a temporary variable that changes every time you store something. It can hold any variable or text. It is the fastest variable. It is mostly useful when you are just manipulating one variable. To use Ans, put an expression or a calculation on a line by itself and it will be stored to Ans.

Hints:
1. One of the best places to use Ans is for reading keypresses.
2. If you are manipulating two variables, it's best to just use the variables.

Example:
getkey
X+(Ans=26)-(Ans=24→X

Reading Keypresses *getKey*

If you want to have the user press a key to do something in your program, use the getKey command. getKey returns a number corresponding to the last key pressed. Each key number has two parts: row (1-10) and column (1-6). If no key has been pressed, getKey returns 0. You can press ENTER at any time during program execution to break the program.

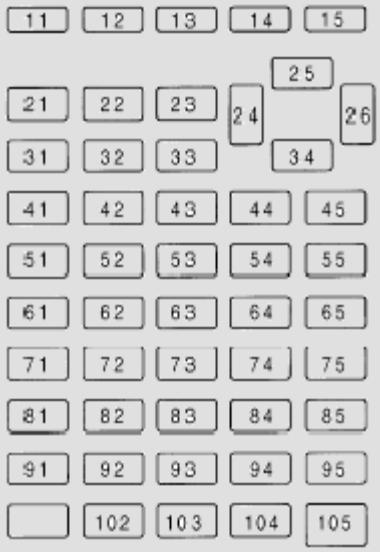
Common getKey Functions:

```
Repeat Ans
getKey→K
end

getKey→K
X+(Ans=26)-(Ans=24→X

Repeat getKey
End
```

Keycode Chart:



Branching *Goto/Lbl*

Definition: Jumping to different parts of a program is sometimes necessary if you don't want every part of a program to run or if you want to skip to a certain part of a program if a certain condition occurs. The commands to do this are Goto (go to) and Lbl (label).

<u>*Goto/Lbl*</u>: In order for Goto and Lbl to work, you need to have both. The label can be one or two characters (A-Z, 0-9, or θ). When Goto is encountered, it notes the label and proceeds to search for it from top to bottom in the program. This can really be slow if the label is deep within the program. It also has the tendency to make your code harder to follow. And, if you use Goto/Lbl incorrectly it can lead to memory leaks. A memory leak is where you exit a loop or a conditional by using Goto. If done enough times, the calculator will run out of memory. So, you should use Goto/Lbl sparingly.

Format:

```
Goto character1(character2)
Lbl character1(character2)
```

Example:

```
Lbl AA
X+1→X
If X<100
Goto AA
```

Subprograms *prgm Return*

Subprograms are programs that are called by other programs to do a particular task. A subprogram is useful for doing repetitive tasks. Instead of having to type the code for a function several times, you just type it once and call the program whenever you want to use it in your program. This makes your program smaller and it keeps it going as fast as possible. It also makes editing and debugging easier. You just have to change the appropriate subprogram instead of going through the entire program looking for the code you want to change. This saves a lot of time and it helps prevent you from accidentally changing other parts of the program.

When you finish your program, you may want to put it together. The problem with using subprograms is that you will have additional programs that are needed to use your program. If you give someone your program you will have to give them your subprograms as well. So, if a subprogram is only used once then you should put it in your parent program. All you have to do is paste the code from the subprogram in place of the program call.

How to Create a Subprogram:

1. Take the code from the parent program and put it in a new program
2. Put a Return command whenever you want to return to the parent program (a Return command isn't needed at the end of a subprogram)

How to Call a Subprogram:

1. Place the cursor in your main program where you want the program to run
2. Put the prgm command on the line
3. Select the program you want from the program list and press ENTER to put it into your program

Hints:

1. Subprograms should be named Zparentn, where parent is the name of the parent program and n is the number (if you have more than one).
2. All variables are global; variables used by one program can be used by another.
3. Labels are local; you can't use Goto in one program to jump to a Lbl in another program.

Example:

```
PROGRAM:ENTER
ClrHome
Repeat K=105
Output(1,1,"Press Enter!
getKey
If Ans=105
prgmZENTER
End

PROGRAM:ZENTER
Pause "U Pressed 105
ClrHome
Return
```

Matrices *[A]-[J]*

Matrices are two-dimensional lists (row x column). They are used for holding lots of information. There are ten built-in matrices ([A]-[J]). A matrix may have up to 99 rows or columns (depending on available memory). There are several commands that you can use with matrices.

Hints:

1. Matrices are often used for setting up a map and checking hit detection.
2. Matrices use more memory than lists or strings, so use them instead, if possible.
3. Comparing elements in a matrix isn't very fast, so store the elements to variables if you are going use them a lot.

Types: dim, Fill, randM, augment, T

<u>*dim*</u>: Used to find the dimensions of a matrix. You can create a matrix by saying how many rows and columns you want and storing it as a matrix. When you first create a matrix all of the elements are zeros. You can also use it to redimension an existing matrix. The elements in the old matrix that are within the new matrix are not changed. The extra elements are filled by zeros. Elements in the old matrix that are outside the new dimensions are deleted.

Example:

```
{3,4→dim([A]
```

<u>*Fill*</u>: Replaces each element in the matrix with the specified value.

Example:

```
Fill(5,[A]
```

<u>*randM*</u>: Creates a random matrix of integers. The seed value stored to rand controls the values of the integers. The default is between negative and positive nine.

Example:

```
randM(3,4→[A]
```

<u>*augment*</u>: Appends one matrix to another as new columns. Both matrices must have the same number of rows.

Example:

```
augment([A],[B]
```

<u>*T*</u>: Returns a matrix in which each element (row, column) is swapped with the corresponding element (column, row) of the matrix.

Example:

```
[A]T
```