

# Coding the Quadratic Formula Using TI84-CE Python

Timothy W. Jones

January 15, 2022

## Abstract

Texas Instruments have added coding in Python to their TI-83 family of calculators. The question this paper attempts to address is why. This investigation starts by considering the programming language of Python and its benefits, especially as contrasted with TI-84 Basic (the standard language for these calculators). It then considers the implementation issues that confront the idea. As an example, Python is highly extensible, but calculators are by their nature highly proprietary, not extensible. And then there is the interface with its other products Smartview and Connect. These are designed to aid teachers and programmers respectively by porting calculator features to PC programs. Does Python inter-phase with these? How well? These concerns are motivated and organized by a concrete programming challenge: seek to code the quadratic formula (we'll define what that means) in Python and attempt to port it to a calculator – as easily as possible, if possible, noting issues and problems as we go along.

## Introduction

I suspect teachers of high school algebra classes were shocked to see Python on student calculators. What on earth could that mean was my initial reaction. I had heard of the programming language Python and occasionally was tempted to try to learn it, but always my particular thought was why bother. I already knew Javascript and TI-84 Basic and that seemed enough for my

needs. That said programming in TI-83 Basic had proven to be frustrating for me several times.

The two most annoying things are TI-Basic does not implement functions and variable names are limited to one capital letter. It is difficult, under these constraints to structure code, especially when, as a teacher, you should show good programming styles. Knowing Javascript well made this annoyance pronounced; I knew structuring my programs was possible in a different language, like Javascript, but alas not the language of these TI calculators. So I was open to the idea of TI-84 Python, even more would I be open to TI-84 Javascript!

If all of the above sounds similar to your experiences, you will find it hardening to know that Python has some very nice features. In particular it kind of forces good programming structure. It forces forces coders to indent lines; in fact, it delimits using indentation! That is its most salient feature. It also, as you would expect, supports functions and varyingly long variable naming. It is a robust language, comparable to Javascript or C. The latter is suggested by its use of an import idea.

Smartview and Connect do support Python, but not as strongly as these support TI-Basic. One can't edit Python code in Connect and port it for testing to a physical calculator, for example. This inter-phase is the standard mode for TI-Basic programs. To get Python programs into Smartview from a physical calculator, attached via a usb cord, is not as clear and clean as doing the same with a TI-Basic program. It can be done.

Enough of coming attractions. I will show issues, constraints, beauties, and annoyances by way of a programming challenge: code the quadratic formula (QF) in Python on a TI-84 CE with Python calculator using, as possible conveniences, Connect and Smartview. I've done the same in classic TI-Basic, so a compare and contrast opportunities will arise. First, what does it mean exactly to code the QF?

## QF: The discriminant

Let's start with something easy. Prompt for the coefficients of the generic  $Ax^2 + Bx + C$  quadratic, crunch the discriminant,  $B^2 - 4AC$ , and display the result. Smartview can do both TI-Basic and Python programs easily, in theory. I say theoretically because it took me a few seconds to do it in Basic and half an hour to do it in Python.

I actually gave up trying to input the simple Python code using the calculator's editor. The problem is one has to step through all characters of Python and you must constantly figure out whether you are in alpha mode lower case, alpha mode upper case, or non-alpha mode regular. So trying to type  $A = \text{int}(\text{input}("A = "))$  is a real annoying challenge. Granted one can type this in or one can navigate the menu system and find *int* and *input*, but then you might be in insert mode or type over mode – in addition to lower case, upper case, regular case modes just mentioned.

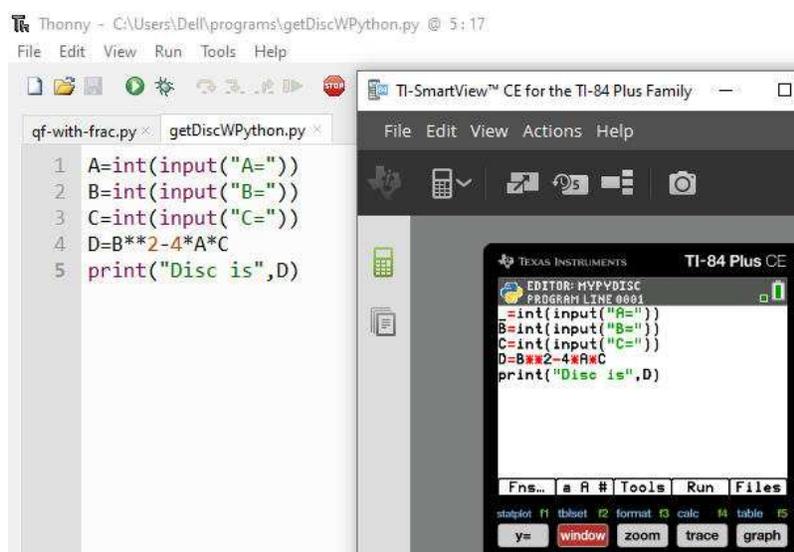


Figure 1: Use a Python editor to make the code for the calculator.

Immediately one senses why designers made TI-Basic so constrained. Reserved words like *disp* and *prompt* delete in one keystroke and are treated as units: no ambiguity in their creation, you must drill into the program menu system to create them. There is no case sensitivity for user created words as there is just one case: upper.

But: you can use an editor to create Python code and bring it easily into Smartview. Thonny is a nice, free editor. Figure 1 shows how I ended up creating the program successfully. Note the missing capital *A* in the calculator's editor screen shows the problem of ambiguous character entry modes. I think it's in insert mode, but I'm not sure. Note: the manual for TI-84 Python stresses how to Smartview (and Connect) can inter-phase with a Python environment, as they call it, like Thonny: for good reason. Entering

Python code using the build in editor is best done by those only into serious sadomasticism.



Figure 2: Caption for ti-getDisc



Figure 3: Caption for ti-get3

The code for the basic version is given in Figure 2. Here I use the calculators version of functions. I made a GET3, Figure 3, program and inserted into the GETDISC program. My motivation is that I frequently want to get three variables named A, B, and C and rather than make each afresh for a program, it is good coding practice to make one version and re-use it. Python and TI-84's version of it can do this.

## QF: Cases

We now are in a position to stipulate what we mean by coding the quadratic formula. There are five cases, meaning five types of solutions: a single real and rational solution, two real and rational solutions (that counts as case 2), two real and irrational solutions (involving a simplified radical), for the third case, two complex rational solutions (for the fourth case), and finally two complex irrational (radical) solutions (for the fifth case).

All cases are resolved by an appeal to the discriminant,  $D$ . If  $D = 0$ , Case 1. We must determine if  $D$  is a perfect square, like 4, 9, or 16, for example. This is achieved in Basic by computing  $fPart(sqrt(abs(D)))$ . If



Figure 4: Caption for ti-get-decimal-part-of-d

this is 0, meaning the decimal part is 0, then  $D$  is a perfect square. Code for Basic is given in Figure 4. TI-84 Python doesn't have the equivalent of this function, but, here it is, you can make your own in a jiffy in Python and Python TI84. That code is shown in Figure 5.

```
9 def getDecimalPart(x):  
10     return x-int(x)
```

Figure 5: Caption for python-get-decimal-part

```

001 prgmGETDISC
002 abs(D)+E
003 √(E)+F
004 fPart(F)+G
005 If (D=0)
006 Then
007 Disp "ONE RAT ROOT"
008 End
009 If (D>0)
010 Then
011 If (G=0)
012 Then
013 Disp "RL PS RAT ROOTS"
014 Else
015 Disp "RL RAD ROOTS √("
016 End
017 End
018 If (D<0)
019 Then
020 If (G=0)
021 Then
022 Disp "CP PS RAT ROOTS"
023 Else
024 Disp "CP RAD ROOTS √("
025 End
026 End

```

Figure 6: Caption for ti-qf-shell

The shell program in both Basic, Figure 6 and Python, Figure 7 are given. Figure 15 (below – way) gives test cases. So, if you want to turn this screed into a tutorial, see if you can make both work with the test cases. Notice how the shell for basic is not indented and is hard to read. Indenting per good good style gives errors. Indenting is forced in Python. The next goal is to fill in the details. Notice we are going for exact solutions with radicals, not just roots in approximate decimal forms; we want reduced fractions with simplified radicals, a more difficult proposition.

```

1 from math import *
2 def getDisc(a,b,c):
3     return b**2-4*a*c
4 def getDecimalPart(x):
5     return x-int(x)
6 def getQf(a,b,c):
7     D = getDisc(a,b,c)
8     E = sqrt(abs(D))
9     F = getDecimalPart(E)
10    if (D==0):
11        print("D is zero, one root")
12    if (D>0):
13        print("D is greater than zero, two real roots")
14        if (F==0):
15            print("Real perfect square, rational roots")
16        else:
17            print("Real radical roots")
18    if (D<0):
19        print("D is less than zero, two complex roots")
20        if (F==0):
21            print("Complex perfect square, rational parts")
22        else:
23            print("Complex radical roots")
24    return
25 getQf(3,5,7)

```

Figure 7: Caption for python-shell

## QF: Central peeves

Before filling out the details for the programs, here is a list of peeves. TI84 Python does not have a GCD function. Regular Python does. It's part of the standard math functions that one imports, see Figure 8. One can make, once again, a GCD functions using a nice recursive function; it's an implementation of the Euclidean algorithm. The GCD function is necessary to reduce fractions – mentioned in the various cases. This is shown by way of a screen capture using Connect, Figures 9; Figure 10 shows some of TI-84 Python's math functions – no GCD. The recursive function works on the calculator – we've got a GCD, GD it. Just in case readers are wondering: there are 59 math functions listed at Python's wiki, there are 22 on the TI-84 version of Python.

`math.gcd(*integers)`

Return the greatest common divisor of the specified integer arguments. If any of the arguments is nonzero, then the returned value is the largest positive integer that is a divisor of all arguments. If all arguments are zero, then the returned value is 0. `gcd()` without arguments returns 0.

*New in version 3.5.*

*Changed in version 3.9:* Added support for an arbitrary number of arguments. Formerly, only two arguments were supported.

Figure 8: Caption for python-has-gcd



```
EDITOR: MYGCD
PROGRAM LINE 0001
from math import *
def myGCD(x,y):
    if y==0:
        return x
    r=int(fmod(x,y))
    return myGCD(y,r)
```

Figure 9: Caption for python-my-gcd



```
EDITOR: MYGCD
math module
Math Const Trig
2:fabs()
3:sqrt()
4:exp()
5:pow(x,y)
6:log(x,base)
7:fmod(x,y)
8:ceil()
9:floor()
0:trunc()
A:frexp()
Esc |Modul
```

Figure 10: Caption for ti-python-no-gcd

Another function is required to simplify square roots, to pull out any perfect squares. Here Python shines and in Basic we are forced to make an insert of code – if we wish to hide functionality, per good coding practice. The Basic code is given in Figure 11 and Figure 12 gives the Python function. One is forced in the Basic version to use global variables, the H and J – very inconvenient. The Basic is a stand alone version. Test it with 4, 16, and 500. Note the unicode in the Python version; these give a plus, minus symbol  $\pm$  (Latex does it too) and the square root symbol ( $\sqrt{2}$ ). Former yes in TI84 version of Python, latter no.

Regular Python does then support unicode characters. As mentioned, we need a square root symbol and Thonny and regular Python delivers; the TI84

```

001  Disp "√(D) "
002  Prompt D
003  abs(D)→E
004  √(E)→F
005  fPart(F)→G
006  iPart(F)→M
007  For(X,1,M)
008  If (fPart(E/X2)=0)
009  Then
010  X→H
011  End
012  End
013  E/H2→J
014  If (J=1)
015  Then
016  Disp H, "PS"
017  Else
018  Disp H, "√(", J
019  End

```

Figure 11: Caption for ti-basic-simplify-radical

and its Python does not. In contrast TI-Basic does via navigation into its menu system: the second test key (see Figure 11). There is a *char* function in regular Python and a *chr* function in TI84 Python, but some characters are generated and some aren't. There appears to be no clear documentation to help a programmer. Yet TI84 Python in its example programs uses `\n` which strikes me as awkward – hailing back to the C programming language.

In general, the import feature in TI84 Python brings in proprietary TI specific modules with sometimes odd naming conventions. This means that you have to be careful in making code in Thonny that you hope will run in your calculator. The manual on TI84 mentions this. What solves this situation is to make Connect support an editor for TI84 Python, something it does not do.

Perhaps here is as a good a place as any to mention that getting programs to work on calculators is generally a silly endeavor, unless you need the portability. In an academic setting, it is good to have a single portable, affordable platform and that's the TI84 CE with Python's main selling point. My usual teaching modality is to code in TI-Basic using Smartview and have

```

5 def getSimplifyRadical(a,d): # called with abs(D) and 2*a
6     H=1
7     for j in range(1,a):
8         if (a/j**2-int(a/j**2)==0):
9             H=j
10            J = a/H**2
11            if (J==1):
12                return getFrac(int(H),d) # drops the sqrt of 1
13            else:
14                return "\u00B1" + "(" + getFrac(int(H),d) + ")" + "\u221A" + str(int(J))

```

Figure 12: Caption for python-simplify-radical

students copy what I do. If I can't code Python in the built in editor, but must go to Thonny (likely not on my classroom's teachers PC), then its potential as a vehicle for teaching Python programming is pretty much zapped from the get-go. But maybe students at home, can use Thonny with youtube videos and enjoy porting it to their calculators for use during tests in their math classes! In this regard, the Python book *Doing Math With Python* tells the back-story of this saga: Python has lots of imports like sympy that do symbolic math, graphics, you name it. This could spell out doom for TI calculators soon! A motive is brewing in my mind for the question why TI Python?

```

def getFrac(a,b):
    c=a/gcd(abs(a),abs(b))
    d=b/gcd(abs(a),abs(b))
    if d!=1:
        return str(int(c))+"/"+str(int(d))
    else:
        return str(int(c))

```

Figure 13: Caption for python-get-frac

Finally, within this category of pet peeves, readers may have noticed a call to *getFrac* in the last Python code. The function called is given in Figure 13. Basic does allow an in-line *to frac* conversion off its math key, Figure 14. But this proves difficult to use in programs; we want a string to concatenate with other strings. This *getFrac* function in turn forced the creation of *myGCD* – one divides out of the numerator and denominator the GCD of the original fraction's versions of these. Excel, regular Python, and TI84's Python do not deliver such a function with two arguments – I had to make this function. There I'm done with irritating things.



Figure 14: Caption for ti-frac

The beauty is you can get complete solutions to quadratics with this calculator, Python, and Excel and it is a great challenge for students.

## QF: Complete solutions

ONEROOT	$X^2 - 2X + 1$	1
RPS	$X^2 + 5X + 6$	-2, -3
RPS	$20X^2 - 23X + 6$	3/4, 2/5
RPS	$4X^2 + 12X - 16$	1, -4
RSQ	$X^2 - 4X + 2$	$2 + \sqrt{2}, 2 - \sqrt{2}$
RSQ	$9X^2 - 30X + 18$	$5/3 \pm \sqrt{7}/3$
IPS	$X^2 - 4X + 8$	$2 + 2i, 2 - 2i$
IPS	$9X^2 - 30X + 34$	$5/3 \pm i$
ISQ	$X^2 - 4X + 6$	$2 + i\sqrt{2}, 2 - i\sqrt{2}$
ISQ	$9X^2 - 30X + 32$	$5/3 \pm i\sqrt{7}/3$

```

40
47  return
48  getQf(1,-2,1)
49  getQf(1,5,6)
50  getQf(20,-23,6)
51  getQf(4,12,-16)
52  getQf(1,-4,2)
53  getQf(9,-30,18)
54  getQf(1,-4,8)
55  getQf(9,-30,34)
56  getQf(1,-4,6)
57  getQf(9,-30,32)

```

Figure 15: Caption for test-quadratics-and-thonny-calls

Drum roll. Figure 15 has the 10 test cases with their sets of three coefficients being called by the function `getQf`. In Thonny these give the correct roots with radicals, Figure 16. As mentioned, TI84 Python can't do these radical signs. So we amend the code as shown in Figure 17: the program I'm going to load into my physical TI84 with Python calculator via Connect. In a moment the results of that trial. Figure 18, captured with Connect, shows success. Note I checked carefully that the name of the square root function was the same in TI84's version.

Using Basic, there is no way to call a function several times without a lot of troublesome work: a for loop that reassigns fixed global variables and reruns a program stored separately. That works, but ugh!

```
>>> %Run qf-with-frac.py
1
-2
-3
3/4
2/5
1
-4
2±(1)√2
5/3±(1/3)√7
2+i2
2-i2
5/3+i1
5/3-i1
2±(1)√2i
5/3±(1/3)√7i
```

Figure 16: Caption for thorny-correct-root-finding-with-radical-sign

## Conclusion

Texas Instruments deserves credit for reading the educational tea leaves. If they add the same functionality to Connect as they provide with Basic, they likely will have a good formula that will stave off for a time a likely future where calculators, like slide rules, become at best quaint.

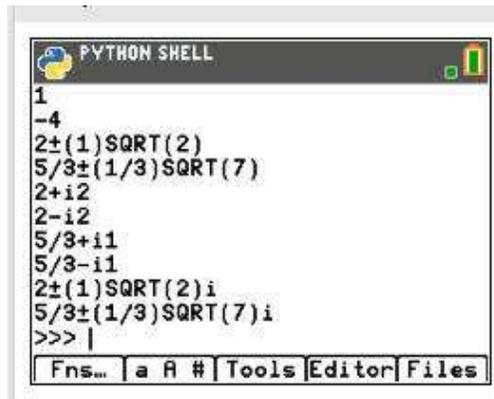
Note the forgo the obvious, programming complete solutions of quadratics (meaning exact solutions too) is a nice challenge. We have used if statements, for loops, recursion, functions, and good programming structure; all was tested with a list of possible types of quadratics. Good classic math crunched well with the latest in technology. I'll make a youtube video later. Not investigated yet is the possible coup de grace for calculators – the graphic components possible with Python! Enough of those tiny screens! And way enough of the pencils. <https://www.youtube.com/watch?v=g3d132La01s>

```

getQFShell.py  qf-with-frac.py  ti-version-getQF.py
1  from math import sqrt
2  def getDisc(a,b,c):
3      return b**2-4*a*c
4  def getDecimalPart(x):
5      return x-int(x)
6  def getGCD(x , y):
7      if y == 0:
8          return x
9      r = int(x % y)
10     return getGCD(y , r)
11 def getFrac(a,b):
12     c=a/getGCD(abs(a),abs(b))
13     d=b/getGCD(abs(a),abs(b))
14     if d!=1:
15         return str(int(c))+"/"+str(int(d))
16     else:
17         return str(int(c))
18 def getSimplifyRadical(a,d): # called with abs(D) and 2*a
19     H=1
20     for j in range(1,a):
21         if (getDecimalPart(a/j**2)==0):
22             H=j
23     J = a/H**2
24     if (J==1):
25         return getFrac(int(H),d) # drops the sqrt of 1
26     else:
27         return "\u0081" + "(" +getFrac(int(H),d) +)" + "SQRT(" + str(int(J)) +)"
28 def getQf(a,b,c):
29     D = getDisc(a,b,c)
30     E = sqrt(abs(D))
31     F = getDecimalPart(E)
32     if (D==0):
33         print(getFrac(-b,2*a))
34     if (D<0):
35         if (F==0):
36             print(getFrac(-b,2*a)+"i"+getFrac(int(E),2*a))
37             print(getFrac(-b,2*a)+"-i"+getFrac(int(E),2*a))
38         else:
39             print(getFrac(-b,2*a) + getSimplifyRadical(abs(D),2*a) +"i")
40     if (D>0):
41         if (F==0):
42             print(getFrac(-b+int(E),2*a))
43             print(getFrac(-b-int(E),2*a)) #call with F and test F==0, F!=0
44         else:
45             print(getFrac(-b,2*a) + getSimplifyRadical(D,2*a))
46     return

```

Figure 17: Caption for ti-qf-code-all-lines



```
PYTHON SHELL
1
-4
2±(1)SQRT(2)
5/3±(1/3)SQRT(7)
2+i2
2-i2
5/3+i1
5/3-i1
2±(1)SQRT(2)i
5/3±(1/3)SQRT(7)i
>>> |
Fns... | a A # | Tools | Editor | Files
```

Figure 18: Caption for connect-add-file-from-computer-part3-success