Coding the Quadratic Formula Using TI84-CE Python

Timothy W. Jones

January 26, 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-83 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-83 Basic and that seemed enough for my needs. That said programming in TI-83 Basic (henceforth just TI-Basic, if I remember) 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 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 (henceforth TI-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 heartening to know that Python has some very nice features. In particular it kind of forces good programming structure. It forces coders to indent lines; in fact, it delimits using indention! 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.

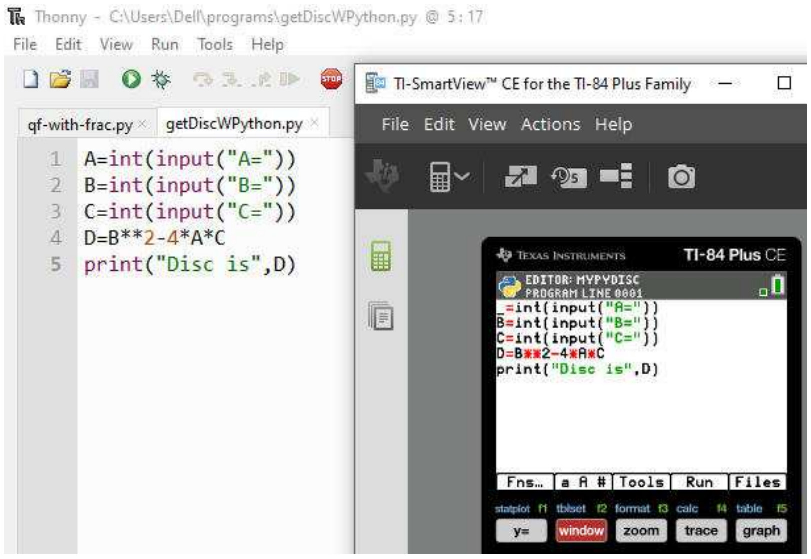
Smartview and Connect do support Python, but not as strongly as these support TI-Basic. One can't edit Python code in Connect and port it for testing to a physical calculator, for example. This inter-phase is the standard mode for TI-Basic programs. To get Python programs into Smartview from a physical 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 TI-Basic, so 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 TI-Python programs easily, in theory. I say in theory 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 $A = int(input(\text{“}A = \text{”}))$ 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 the lower case, upper case, and regular modes just mentioned.


Thonny - C:\Users\Dell\programs\getDiscWPython.py @ 5 : 17
File Edit View Run Tools Help
qf-with-frac.py
getDiscWPython.py
TI-SmartView™ CE for the TI-84 Plus Family
File Edit View Actions Help
Texas Instruments
TI-84 Plus CE
EDITOR: MYPYDISC
PROGRAM LINE 0001
Fns… a A # Tools Run Files
statplot f1 tblset f2 format f3 calc f4 table f5
y= window zoom trace graph

```
A=int(input("A="))
B=int(input("B="))
C=int(input("C="))
D=B**2-4*A*C
print("Disc is",D)
```

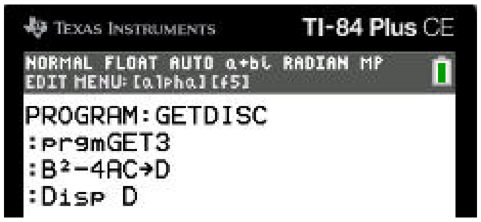
```
_=int(input("A="))
B=int(input("B="))
C=int(input("C="))
D=B**2-4*A*C
print("Disc is",D)
```

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

Immediately one senses (or at least I sense) 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. Did I mention there is a cap locks feature?

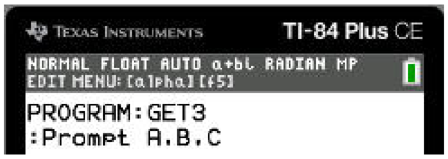
But: you can use an editor to create Python code and bring it into Smartview; its not a drag and drop or a copy and paste; its more a navigate for an hour and hope. Thonny is a nice, free editor. Figure 1 shows how I ended up creating the program successfully. Note the missing capital A in the calculator's 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-Python stresses how 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.

This is a link to python's main page for beginners: Official Python site. It mentions Thonny and gives a link. Here is a link to TI-Python's manual: TI Python manual. Here's a link to me making the code for this document with humorous comments (a bone in my leg annotates code).



```
PROGRAM:GETDISC
:prgmGET3
:B²-4AC→D
:Disp D
```

Figure 2: TI-Basic version of get the discriminant, uses an insert.



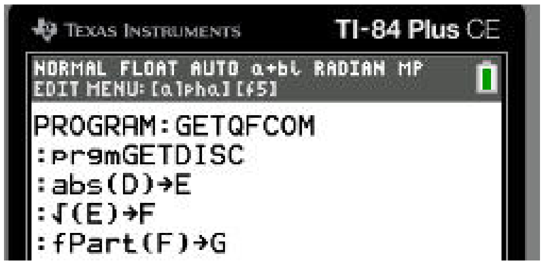
```
PROGRAM:GET3
:Prompt A,B,C
```

Figure 3: TI-Basic has limited function like structures – inserts of code.

The code for the basic version is given in Figure 2. Here I use the calculators version of functions (more like inserts). 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 instance 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 more elegantly, correctly you could say, with functions.

QF: Cases

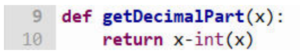
We now are in a position to stipulate what we mean by coding the quadratic formula. There are five cases, meaning five types of solutions: a single real and rational solution (case one), two real and rational solutions (case two), two real and irrational solutions (involving a simplified radical) for the third case, two complex rational solutions for the fourth case, and finally two complex irrational (radical) solutions for the fifth case. See Figure 15 for examples of each of these cases.



```
PROGRAM:GETQFCOM
:prgmGETDISC
:abs(D)→E
:√(E)→F
:fPart(F)→G
```

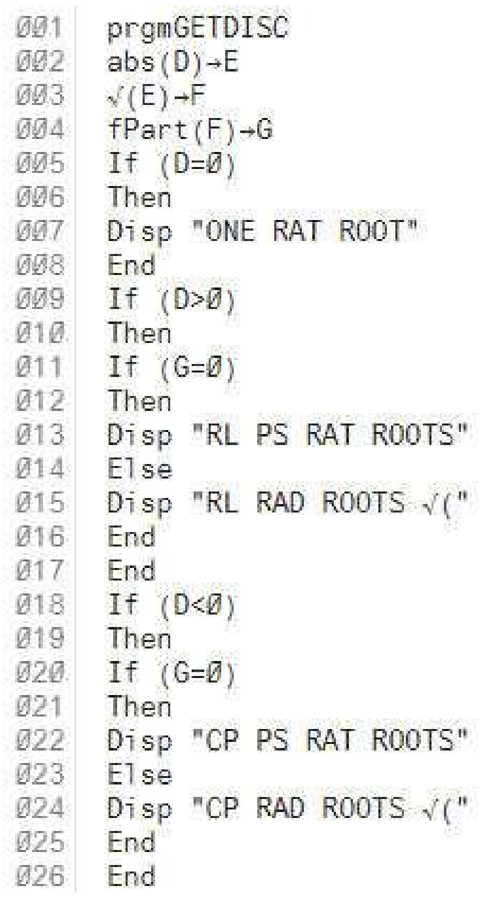
Figure 4: TI-Basic code that gives the decimal part of D, the discriminant.

All cases are resolved by an appeal to the discriminant, D. If $D = 0$, Case 1. For the other cases, 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 this is 0, meaning the decimal part is 0, then D is a perfect square. Code for TI-Basic is given in Figure 4. TI-Python doesn't have the equivalent of the fPart function (floating or decimal part of a real), but, here it is, you can make your own in a jiffy in Python and TI-Python. That code is shown in Figure 5.



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

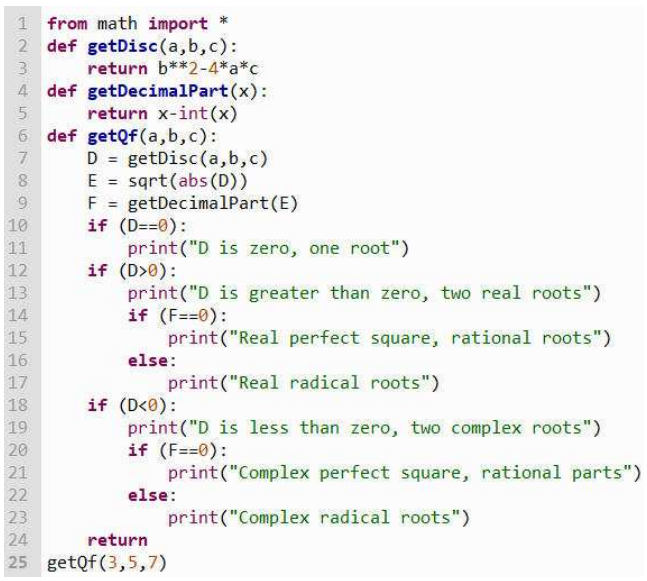
Figure 5: TI-Python function that works like TI-Basic's fPart.



```
prgmGETDISC
abs(D)→E
√(E)→F
fPart(F)→G
If (D=0)
Then
Disp "ONE RAT ROOT"
End
If (D>0)
Then
If (G=0)
Then
Disp "RL PS RAT ROOTS"
Else
Disp "RL RAD ROOTS √("
End
End
If (D<0)
Then
If (G=0)
Then
Disp "CP PS RAT ROOTS"
Else
Disp "CP RAD ROOTS √("
End
End
```

Figure 6: Shell for TI-Basic QF program.

The shell program in both TI-Basic, Figure 6 and TI-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 programming style gives errors in TI-Basic. 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.

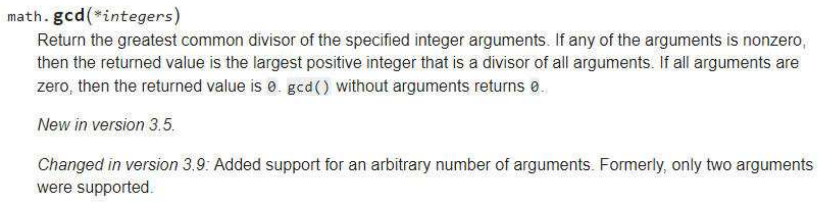


```
from math import *
def getDisc(a,b,c):
    return b**2-4*a*c
def getDecimalPart(x):
    return x-int(x)
def getQf(a,b,c):
    D = getDisc(a,b,c)
    E = sqrt(abs(D))
    F = getDecimalPart(E)
    if (D==0):
        print("D is zero, one root")
    if (D>0):
        print("D is greater than zero, two real roots")
        if (F==0):
            print("Real perfect square, rational roots")
        else:
            print("Real radical roots")
    if (D<0):
        print("D is less than zero, two complex roots")
        if (F==0):
            print("Complex perfect square, rational parts")
        else:
            print("Complex radical roots")
    return
getQf(3,5,7)
```

Figure 7: Shell for TI-Python QF program.

QF: Central peeves

Before filling out the details for the programs, here is a list of peeves. TI-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. Figure 9 shows the function I needed to make; Connect snapped this screen from my physical calculator. The GCD function is necessary to reduce fractions – mentioned in the various cases. Figure 10 shows some of TI-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 Pythons Wiki, there are 22 on the TI-84 version of Python. Not to be overly erudite, TI-Python's manual tells us that they are implementing a small version of Python called Circuit Python.



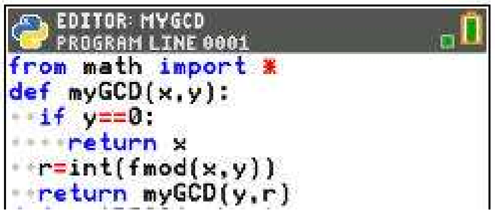
math.**gcd**(**integers*)

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.

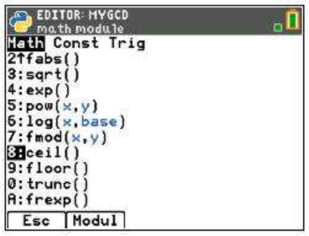
Changed in version 3.9: Added support for an arbitrary number of arguments. Formerly, only two arguments were supported.

Figure 8: Wiki entry for GCD, a standard Python math function.



```
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)
```

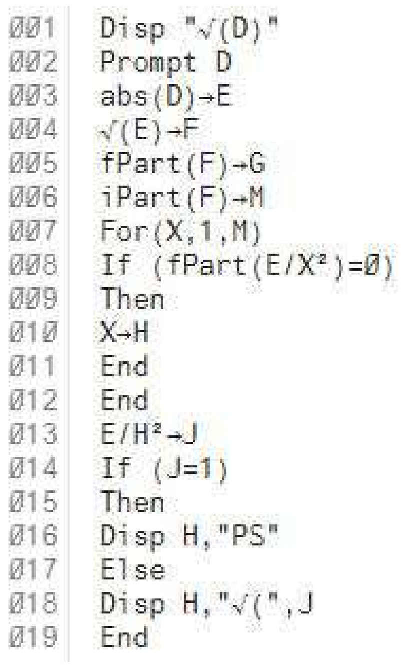
Figure 9: A function in TI-Python providing the GCD function.



```
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: A partial list of TI-Python math functions; 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. We did this hiding with GET3 in the getDISC program earlier. The Basic code is given in Figure 11 and Figure 12 gives the Python function. One is forced in the Basic program (Figure 11) to use global variables, the H and J – very inconvenient. The Basic is a stand alone version. Test it (maybe understand it!) with 4, 16, and 500. Note the unicode in the Python version; these give a plus, minus symbol ± (Latex does it too) and the square root symbol ($\sqrt{2}$). Former yes in TI-Python, latter no.

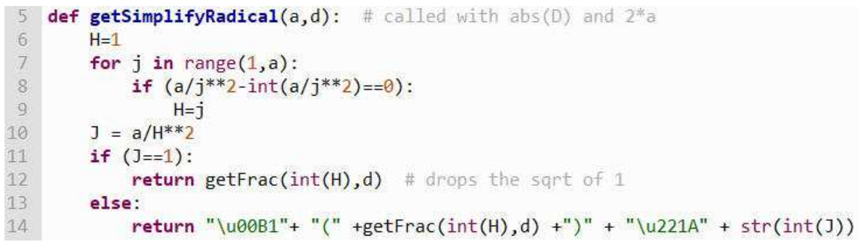


```
001  Disp "√(D)"
002  Prompt D
003  abs(D)→E
004  √(E)→F
005  fPart(F)→G
006  iPart(F)→M
007  For(X,1,M)
008  If (fPart(E/X²)=0)
009  Then
010  X→H
011  End
012  End
013  E/H²→J
014  If (J=1)
015  Then
016  Disp H,"PS"
017  Else
018  Disp H,"√(",J
019  End
```

Figure 11: TI-Basic stand alone simplify radical program.

Regular Python does support, as one would certainly expect, unicode characters – all of them. As mentioned, we need a square root symbol and Thonny and regular Python delivers; TI-Python does not. In contrast TI-Basic does via navigation into its menu system: the second test key (see Figure 11, line 018). There is a *char* function in regular Python and a *chr* function in TI84 Python, but some characters are generated and some aren't. Repetition means annoyance; I'll try to stop. There appears to be no clear documentation to help a programmer (cf. earlier link to TI-Python's manual, nada there). Yet TI-Python in its example programs uses `\n` which strikes me as awkward – hailing back to the C programming language. But regular Python has the same conventions. In contrast Javascript uses a write and writeln for the same functionality – carriage return idea. Backslash n creeps me out – there I said it.

In general, the import feature in TI-Python brings in proprietary TI specific modules with sometimes odd (meaning non-pure-Python) 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 TI-Python



```
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))
```

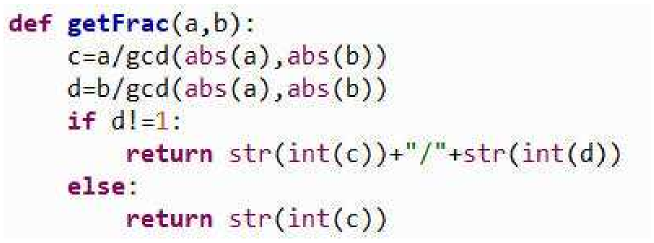
Figure 12: TI-Python simplify radical function.

mentions this. Recall in the abstract I mentioned the inherent problem of porting an expansion friendly Python into an inherently proprietary world of a calculator. Graphics are especially proprietary. What solves this situation is to make Connect support an editor for TI-Python, something it does not do. One could predict a new release of Connect will do this and maybe they will force a new calculator purchase as well! Hm?!

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, if it has one! My usual teaching modality is to code in TI-Basic using Smartview and have students copy what I do. If I can't code Python in the built in editor (Smartview is just a calculator simulator; i.e. no help), but must go to Thonny (likely not on my classroom teacher's 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 and port math programs to their calculators via Connect (a free download), and use the programs during tests in their math classes! That's good for me.

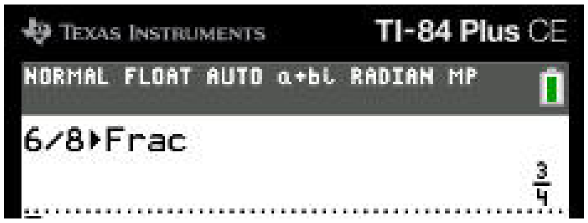
In this regard, the Python book *Doing Math With Python* tells the backstory of this saga: Python has lots of imports like sympy that do symbolic math, graphics, you name it. This could spell out doom for TI calculators soon! A motive is brewing in my mind for the question why TI-Python? But I envision math classes being taught with classroom computers and in this regard I might be out of the mainstream – keep doing math as it was done in 1863 with pencil and paper most teachers say. I mentioned this was a hidden screed.

Finally, within this category of pet peeves (awful unwanted guests?), readers may have noticed a call to *getFrac* in the last Python code, Figure 12, line



```
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: Function getFrac implemented in Python.

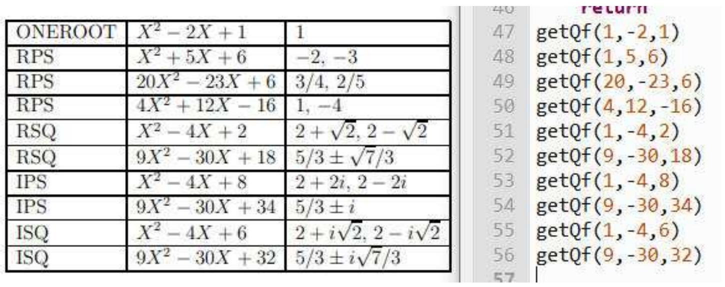


```
6/8▸Frac
                                   3/4
```

Figure 14: TI-Basic has a built in, non-programmatic converter to a fraction.

012. 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, Figure 12, line 014. 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, TI-Python, TI-Basic, not even Javascript provides such a function with two arguments – I had to make it myself. There I'm done with irritating things. Annoying house guests purged, let's go on, shall we? (Line from Blade Runner – shows my inter-reaction with TI help desk, as does Barton Fink hall scene with John Goodman; I had the code in my policy case! Barton Fink, Blade Runner. Turtle code is an interesting feature of Python.)

The beauty is you can get complete solutions to quadratics with this calculator's TI-Python and it is a great math and programming challenge for students (and teachers).



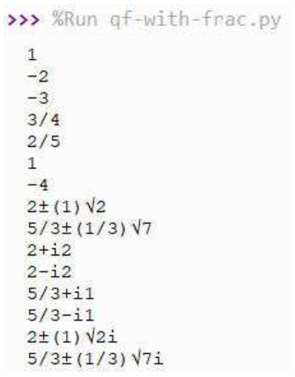
ONEROOT	$X^2 - 2X + 1$	1
RPS	$X^2 + 5X + 6$	$-2, -3$
RPS	$20X^2 - 23X + 6$	$3/4, 2/5$
RPS	$4X^2 + 12X - 16$	$1, -4$
RSQ	$X^2 - 4X + 2$	$2 + \sqrt{2}, 2 - \sqrt{2}$
RSQ	$9X^2 - 30X + 18$	$5/3 \pm \sqrt{7}/3$
IPS	$X^2 - 4X + 8$	$2 + 2i, 2 - 2i$
IPS	$9X^2 - 30X + 34$	$5/3 \pm i$
ISQ	$X^2 - 4X + 6$	$2 +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

```
    return
getQf(1,-2,1)
getQf(1,5,6)
getQf(20,-23,6)
getQf(4,12,-16)
getQf(1,-4,2)
getQf(9,-30,18)
getQf(1,-4,8)
getQf(9,-30,34)
getQf(1,-4,6)
getQf(9,-30,32)
```

Figure 15: Quadratic test cases and Thonny Python calls to getQf function.

QF: Complete solutions

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, TI-Python can't do these radical



```
>>> %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: Regular Python, made in Thonny, output of QF program. All cases correct. My nails shiny.

signs. I'm not mad about that (cf. John Goodman in Barton Fink, honest I'm not). So we amend the code as shown in Figure 17, line 027 (compare Figure 12, line 014) and produce 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 in regular Python was the same in TI-Python. For really obsessive compulsive readers only: notice how I switched out line 8 of Figure 12 and put in a more readable reference to getDecimal, Figure 17, line 21.

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!

Conclusion

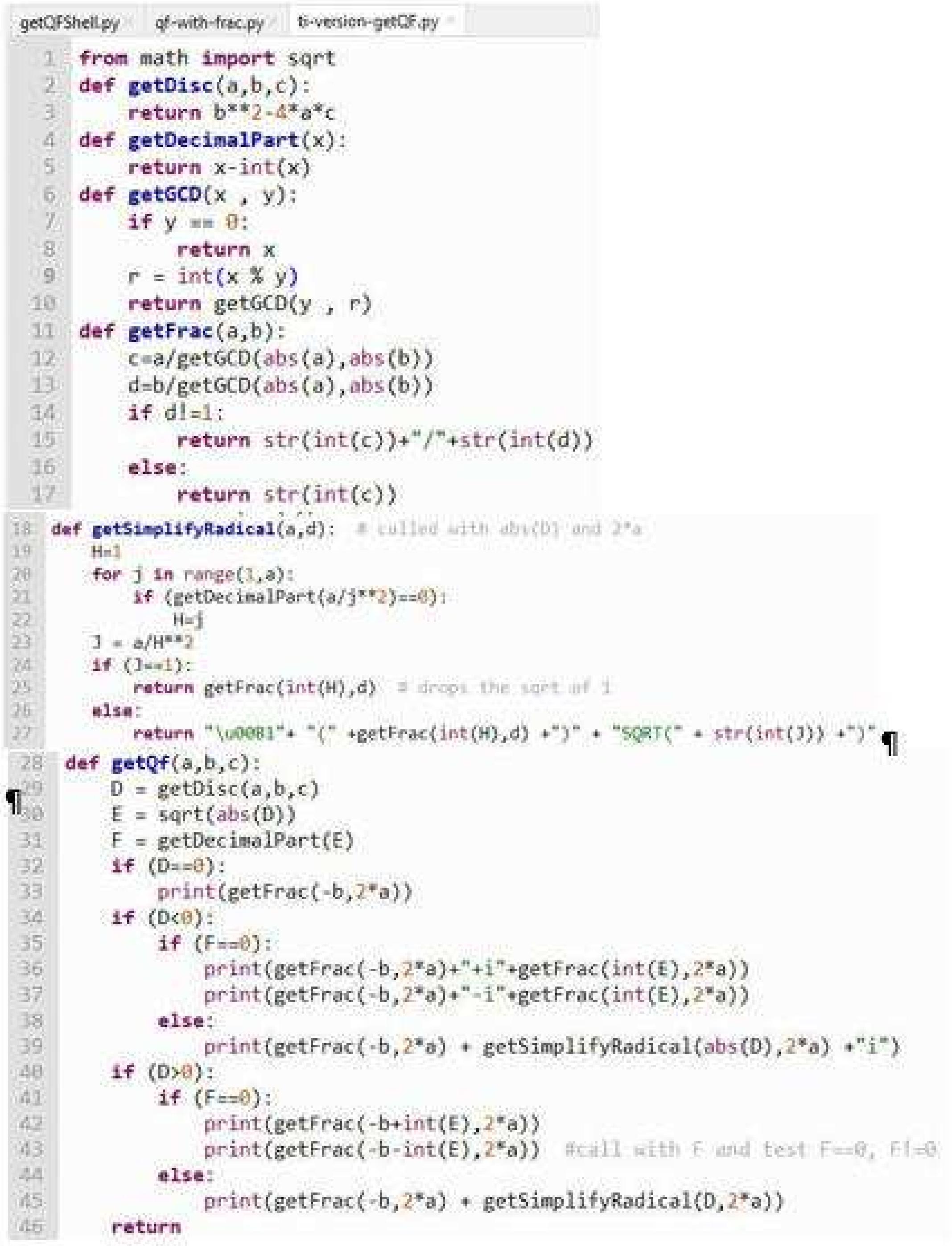
Texas Instruments deserves credit for reading the educational tea leaves. If they add the same Python functionality to Connect as they provide with Basic, they likely will have a good formula that will stave off for a time a likely future where calculators, like slide rules, become at best quaint – if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

Not to forgo the obvious, programming complete solutions of quadratics is a nice challenge. We have used if statements, for loops, recursion, functions, and good programming structure; all were 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; Pencil Trick during TI Board meeting.

As a practical matter, for my fellow teachers, I'm planning on assigning this program as an extra credit project (exemption from midterm) in a college algebra class. We use CANVAS and using it students can insert screen captures from their calculator and Thonny into a word document, convert this document into a pdf file, and upload it into CANVAS. A TI-84 CE with Python coupled with Connect allows for importing Python files created in Thonny and capturing calculator screen shots, so all should be doable, but it is an experiment. In a future draft of this document, the result of that trial.

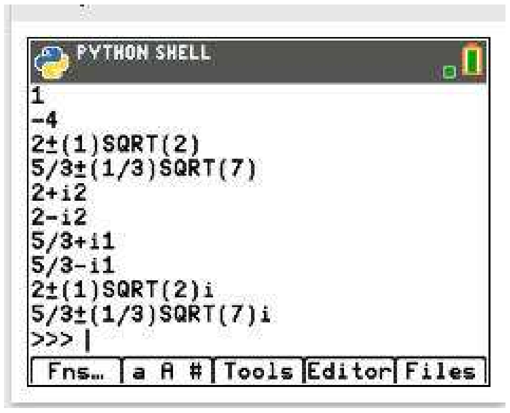
I further plan on proselytizing this to my school's math department. It is the sort of thing all teachers should do. It is a necessary and good upgrade in pedagogy. All students should learn math this way.

Complaints; I teach math, not programming. But please notice how organized, modern, and deep is the understanding of a student who completes such a programming assignment. More than this, with the coding I have done in class with TI-Basic (using Smartview) students are engaged and get a thrill from meeting a challenge of getting technology to do the mindless grunt work of adding and subtracting accurately. They like it and they do not like doing math as it was done in 1714. May regression models be ingrained in all high school students heads – so they will know that equations govern reality, not old people and their old school ways!



```
 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: Regular Python code modified (line 27) for upload to calculator via Connect.



```
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
>>> |
```

Figure 18: The output of QF program on physical calculator. Captured via Connect.