

TI-84 Plus CE Python: Sample Programs

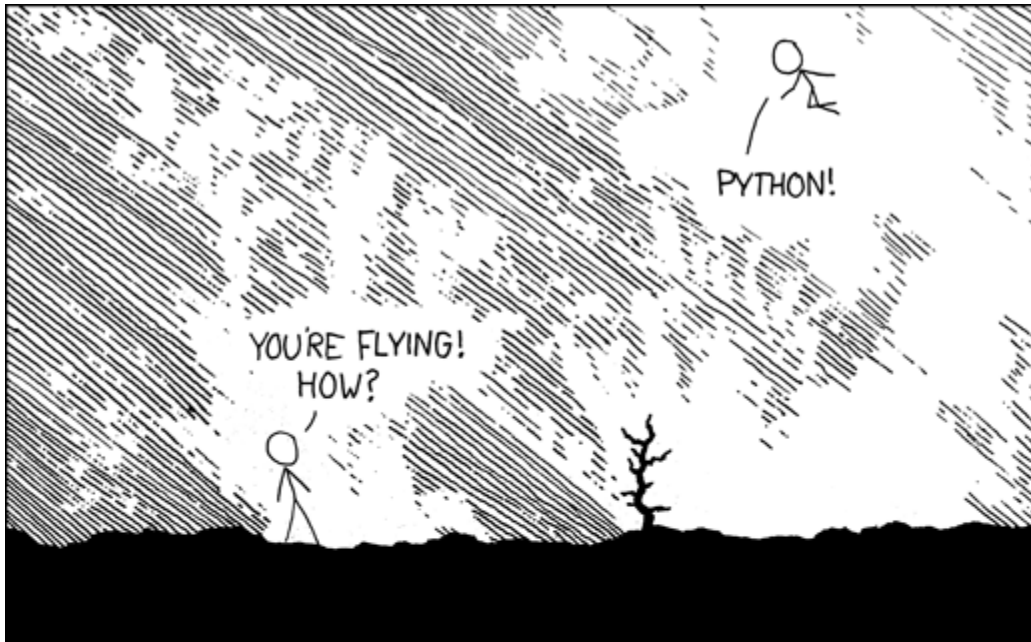
Webinar, Nov 16, 2021

John Hanna, johnhanna@gmail.com

Becky Byer, bbyer31415@gmail.com

with guidance from Steve Phelps

No program is ever finished.



Permanent link to this comic: <https://xkcd.com/353/>

The Big Picture

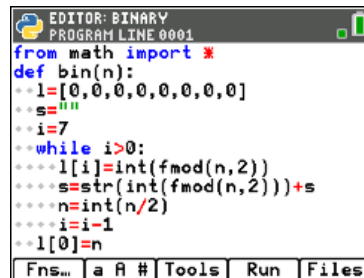
The **TI-84 Plus CE Python Edition** is a *new* graphing calculator that supports Python programming using the 'Python App'. Pressing the **[prgm]** key now provides *two* options:



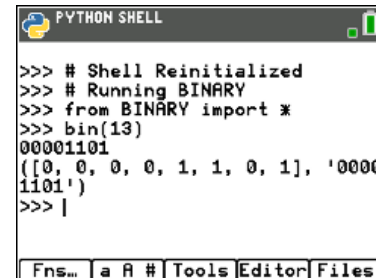
Selecting the **Python App** takes you to the Python programming experience which is a *separate* system from the rest of the graphing calculator. There are three workspaces in the Python App:



File Manager



Editor



Shell

File Manager: where new Python files are created and allows you to **<Manage>** files (copy, rename, delete)

Editor: where Python code is written. **<Tools>** contains editing utilities like Copy line and Paste line.

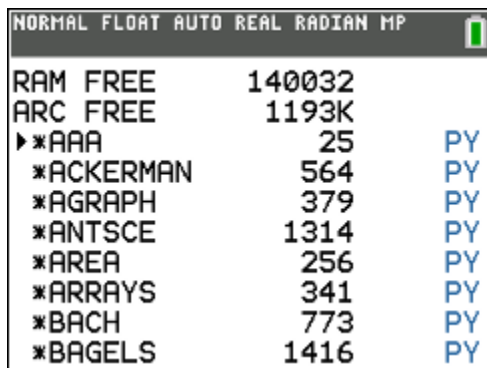
Shell: where Python programs run. Similar to the calculator Home screen, also allows for evaluating expressions and testing short code chunks.

Soft Keys: all three workspaces have 'soft keys' on the bottom of the screen. Use the top row of keypad keys to access these soft keys.

Memory Management

Python files are stored as **AppVars**.

Active (available) files are stored in RAM. Backup files are stored in ARCHIVE (*).

A screenshot of the 'Memory Management' workspace. It shows a table with the following data:

	RAM FREE	140032	
	ARC FREE	1193K	
▶*	AAA	25	PY
*	ACKERMAN	564	PY
*	AGGRAPH	379	PY
*	ANTSCE	1314	PY
*	AREA	256	PY
*	ARRAYS	341	PY
*	BACH	773	PY
*	BAGELS	1416	PY

[quit] the Python App.

Press **[mem] > Memory Management > AppVars**

Python files are indicated by the **PY** on the right.

Press **[enter]** to move a file between RAM and ARCHIVE.

Note the asterisks (*) indicating the archived files.

In the Python App you are limited to 100 Python files and 50k RAM so archiving becomes important. Archived files will not appear in the Python App File Manager.

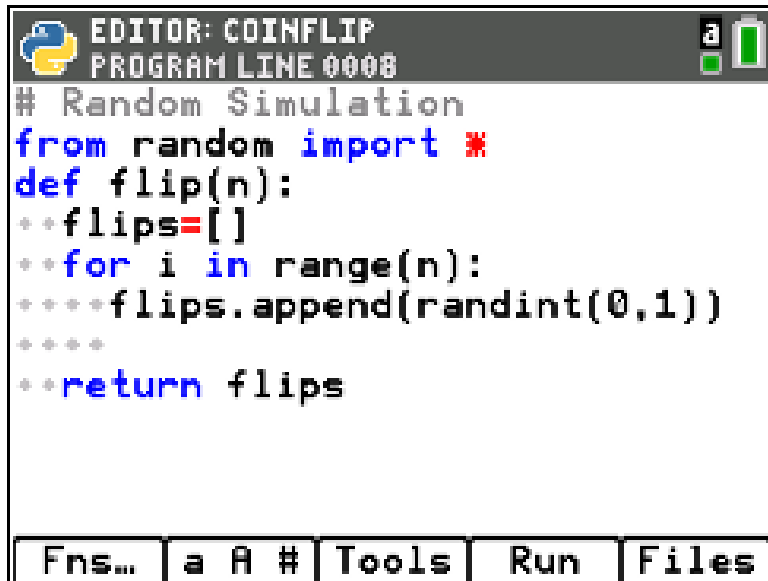
Editing Tips

Use the menus. Use short variable names. Indentation is crucial! Python is case-sensitive: use

[alpha] or **[2nd] [alpha]** and pay attention to the on-screen prompts and cursor for the current state (lower, upper, or numeric). The **[math]** key is a shortcut for **<Fns...> Modul**.

Sample functions and programs

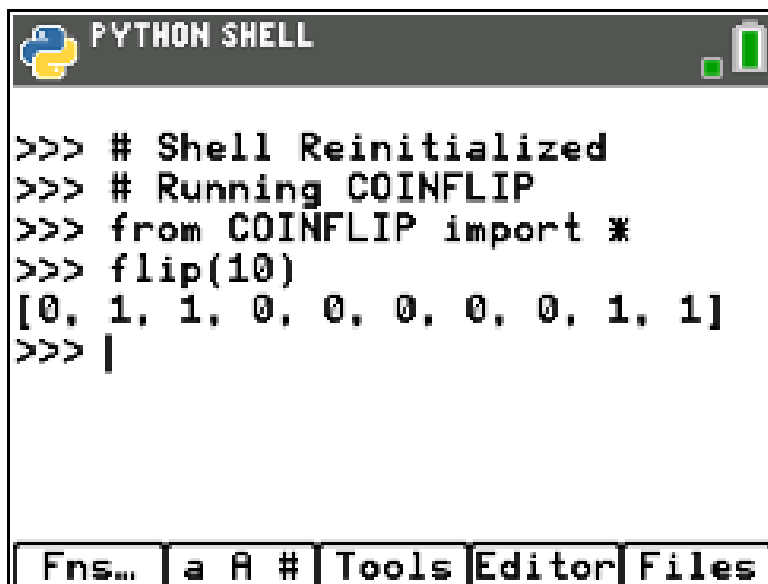
A Coin-Flipping Function



```
EDITOR: COINFLIP
PROGRAM LINE 0008
# Random Simulation
from random import *
def flip(n):
    flips=[]
    for i in range(n):
        flips.append(randint(0,1))
    return flips
```

The screenshot shows a Python IDE window titled "EDITOR: COINFLIP" with a status bar at the bottom containing buttons: "Fns...", "a A #", "Tools", "Run", and "Files". The code defines a function `flip(n)` that simulates `n` coin flips by appending random integers (0 or 1) to a list.

select <Run>... then press [vars] and select `flip(10)` (type the 10)

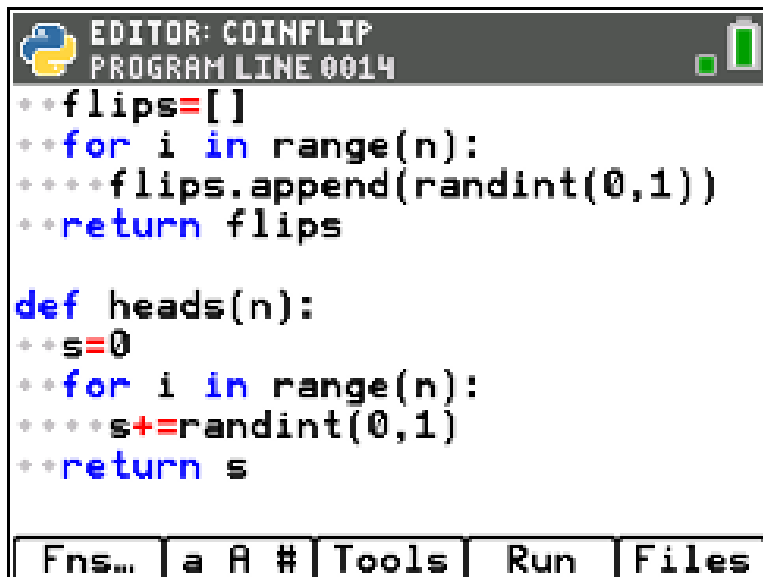


```
PYTHON SHELL
>>> # Shell Reinitialized
>>> # Running COINFLIP
>>> from COINFLIP import *
>>> flip(10)
[0, 1, 1, 0, 0, 0, 0, 0, 1, 1]
>>> |
```

The screenshot shows a Python Shell window titled "PYTHON SHELL" with a status bar at the bottom containing buttons: "Fns...", "a A #", "Tools", "Editor", and "Files". The shell displays the execution of the `flip(10)` function, which returns a list of 10 random integers: `[0, 1, 1, 0, 0, 0, 0, 0, 1, 1]`.

How Many Heads? Function

(note: same file!)

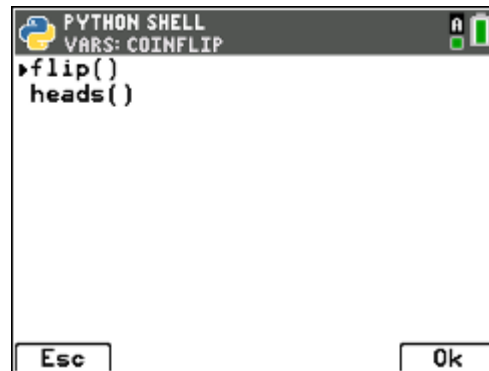


```
EDITOR: COINFLIP
PROGRAM LINE 0014

**flips=[]
**for i in range(n):
***flips.append(randint(0,1))
**return flips

def heads(n):
**s=0
**for i in range(n):
***s+=randint(0,1)
**return s

Fns... | a A # | Tools | Run | Files
```



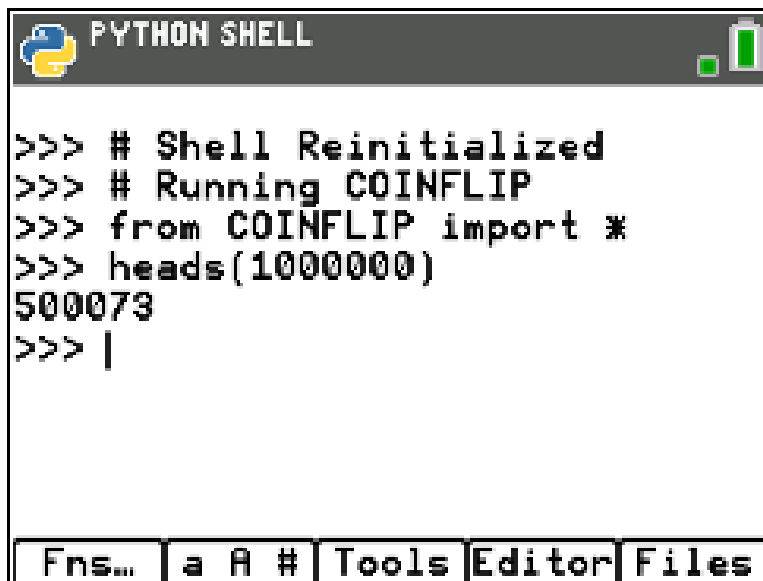
```
PYTHON SHELL
VARS: COINFLIP

> flip()
heads()

Esc | Ok
```

press **[vars]** to see this...

then select **heads()** from the list...

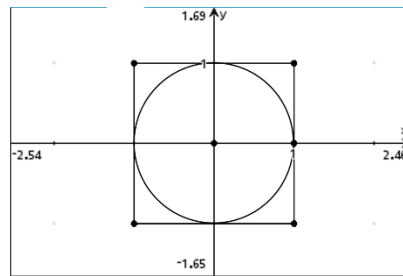


```
PYTHON SHELL

>>> # Shell Reinitialized
>>> # Running COINFLIP
>>> from COINFLIP import *
>>> heads(1000000)
500073
>>> |

Fns... | a A # | Tools | Editor | Files
```

Pi Darts Program



Darts are thrown randomly at the unit circle inscribed in a 2x2 square. Some land inside the circle, others don't. On average, what percent of the darts land inside the circle? How can we use this to compute an estimate for pi?

```
EDITOR: PIDART
PROGRAM LINE 0005
# Random Simulation
from random import *
from ti_system import *
h=t=0
while not escape():
    t+=1
    x=uniform(-1,1)
    y=uniform(-1,1)
    if x**2+y**2<1:
        h+=1
    print(4*h/t)
Fns... | a A # | Tools | Run | Files
```

PIDARTS: A graphical version:

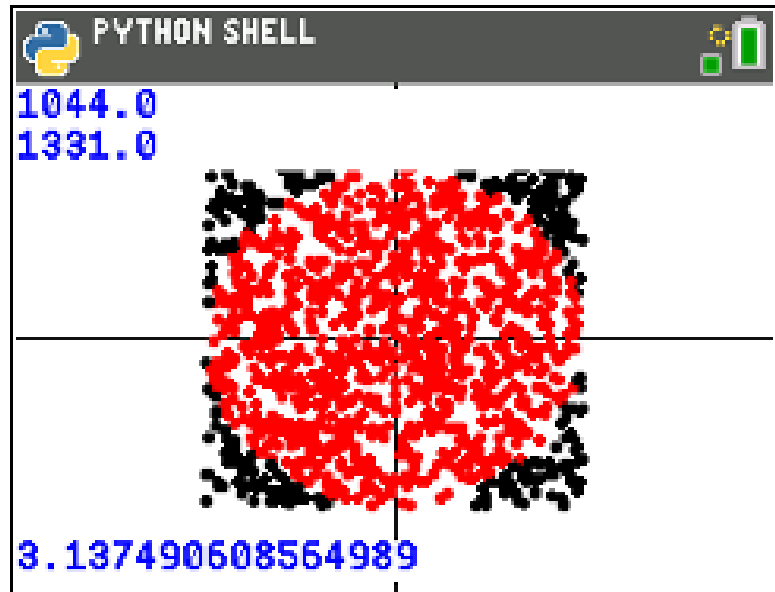
```
from ti_system import *
from random import *
import ti_plotlib as plt
```

```
plt.cls()
plt.window(-2,2,-1.5,1.5)
plt.axes("axes")
```

```
try:
    l=recall_list("DARTS")
    hits=l[0]
    total=l[1]
except:
    hits=0
    total=0
```

```
while not escape():
    total+=1
    x=uniform(-1,1)
    y=uniform(-1,1)
    plt.color(0,0,0)
    if x*x+y*y<=1:
        hits+=1
        plt.color(255,00,0)
        plt.plot(x,y,"o")
    plt.color(0,0,255)
    plt.text_at(2,str(total),"left")
    plt.text_at(1,str(hits),"left")
    plt.text_at(12,str(hits/total*4),"left")
```

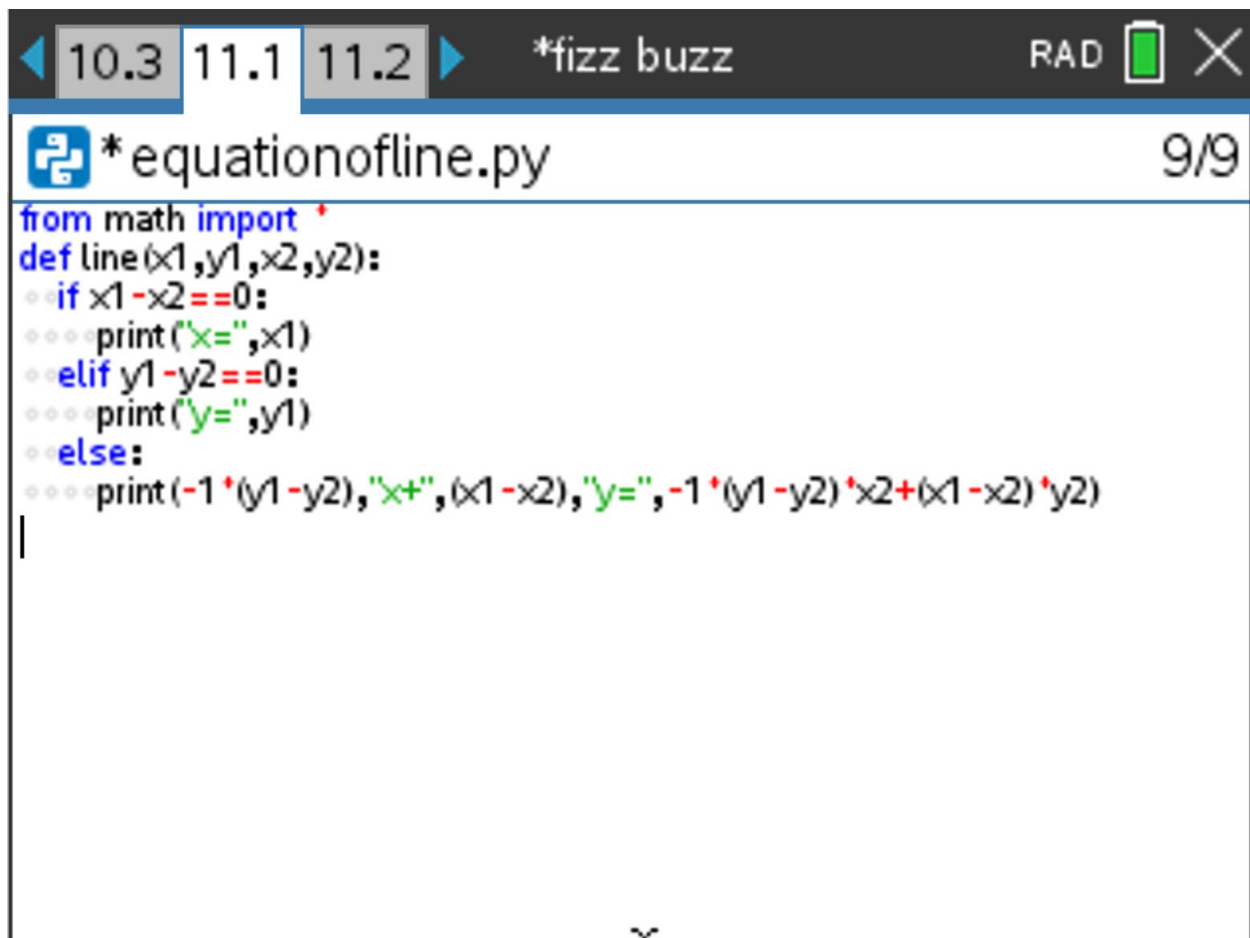
```
plt.show_plot()
store_list("DARTS",[hits,total])
```



Other samples from TI-Nspire CX II

Equation of a Line in Standard Form

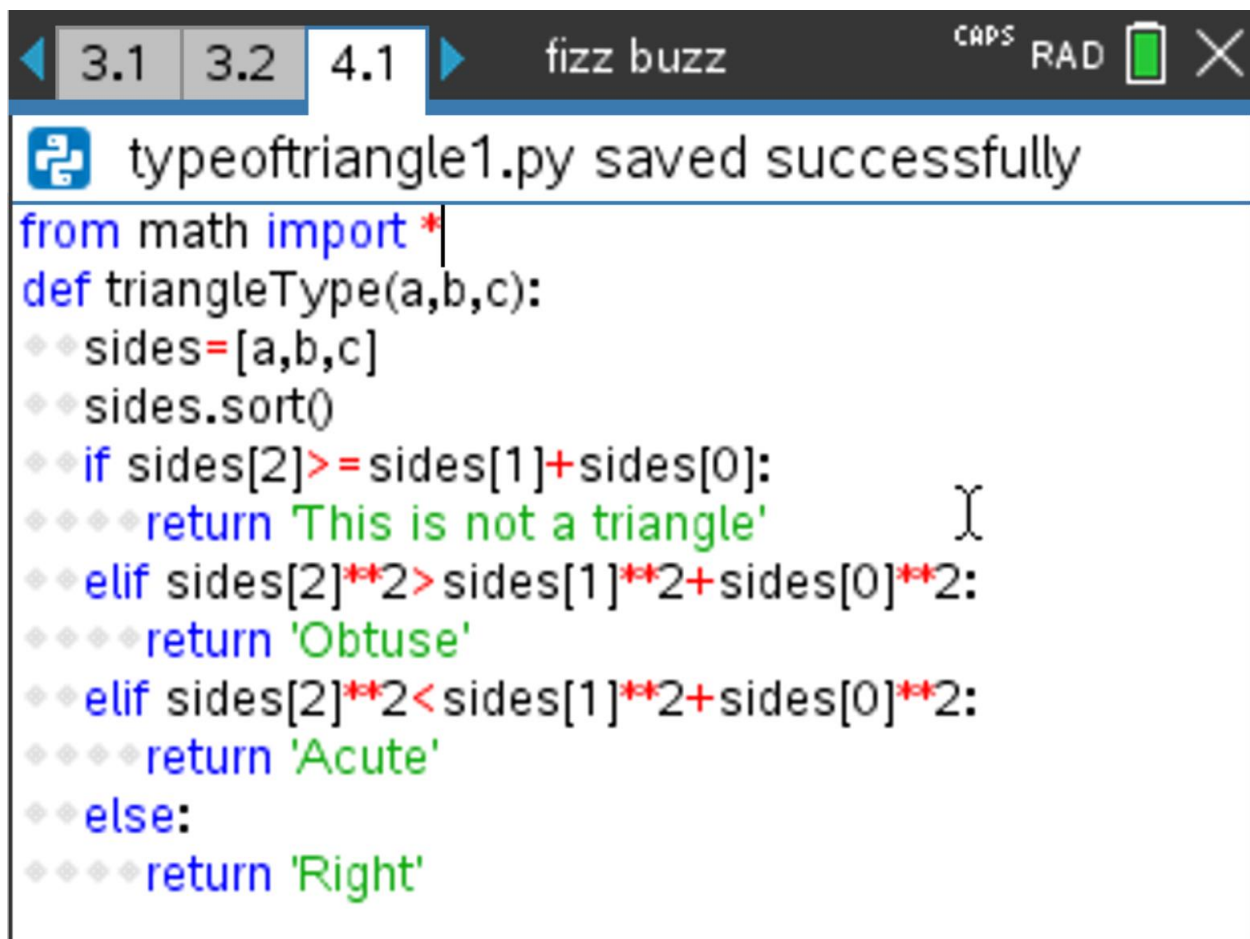
Call the function `line(x1,y1,x2,y2)` which returns the equation of a line that passes through the points $(x1,y1)$ and $(x2,y2)$



The image shows the TI-Nspire CX II interface. At the top, there are navigation buttons for sections 10.3, 11.1, and 11.2, with 11.1 currently selected. To the right of these buttons is a search bar containing the text '*fizz buzz'. Further right are icons for 'RAD' mode, a battery level indicator, and a close button. Below the navigation bar, the title bar of a script window reads '*equationoffline.py' with a page indicator '9/9' on the right. The main area of the window contains a Python script that defines a function 'line' which takes four arguments: x1, y1, x2, and y2. The function uses conditional logic to handle vertical lines (x1 == x2), horizontal lines (y1 == y2), and general cases. For the general case, it calculates the slope and y-intercept to form the equation in standard form: $-1 \cdot (y1 - y2) \cdot x + (x1 - x2) \cdot y = -1 \cdot (y1 - y2) \cdot x2 + (x1 - x2) \cdot y2$.

```
from math import *
def line(x1,y1,x2,y2):
    if x1-x2==0:
        print("x=",x1)
    elif y1-y2==0:
        print("y=",y1)
    else:
        print(-1*(y1-y2),"x+",(x1-x2),"y=", -1*(y1-y2)*x2+(x1-x2)*y2)
```

What Type of Triangle?

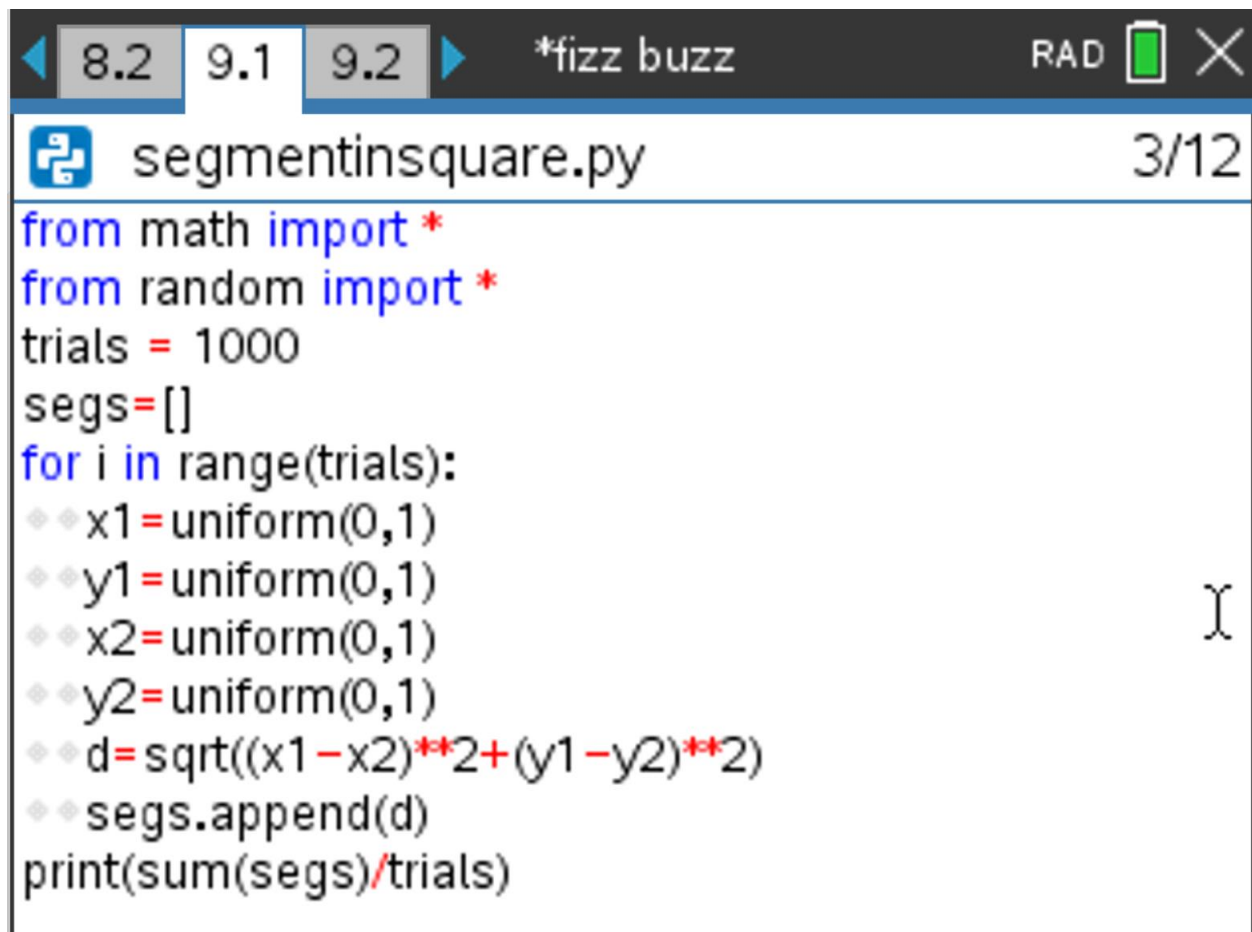


The image shows a Jupyter Notebook window with a dark header bar. On the left, there are three tabs labeled '3.1', '3.2', and '4.1', with '4.1' being the active tab. To the right of the tabs is the text 'fizz buzz'. Further right are icons for 'CAPS', 'RAD', a battery level indicator, and a close button. Below the header bar, a status message reads 'typeoftriangle1.py saved successfully' next to a file icon. The main area of the notebook contains a Python script for determining the type of a triangle. The script imports the 'math' module and defines a function 'triangleType(a,b,c)'. Inside the function, it sorts the sides, then uses conditional logic to return 'This is not a triangle' (if the sum of two sides is less than or equal to the third), 'Obtuse' (if the square of the longest side is greater than the sum of the squares of the other two), 'Acute' (if the square of the longest side is less than the sum of the squares of the other two), and 'Right' (if the square of the longest side is equal to the sum of the squares of the other two). The script is written in a syntax-highlighted style with blue for keywords, green for strings, and red for operators and punctuation.

```
from math import *  
def triangleType(a,b,c):  
    sides=[a,b,c]  
    sides.sort()  
    if sides[2]>=sides[1]+sides[0]:  
        return 'This is not a triangle'  
    elif sides[2]**2>sides[1]**2+sides[0]**2:  
        return 'Obtuse'  
    elif sides[2]**2<sides[1]**2+sides[0]**2:  
        return 'Acute'  
    else:  
        return 'Right'
```


Segments Inside a Square

Draw two random points inside a square of side length 1 and connect them with a segment. What is the average length of the typical segment drawn in this manner?



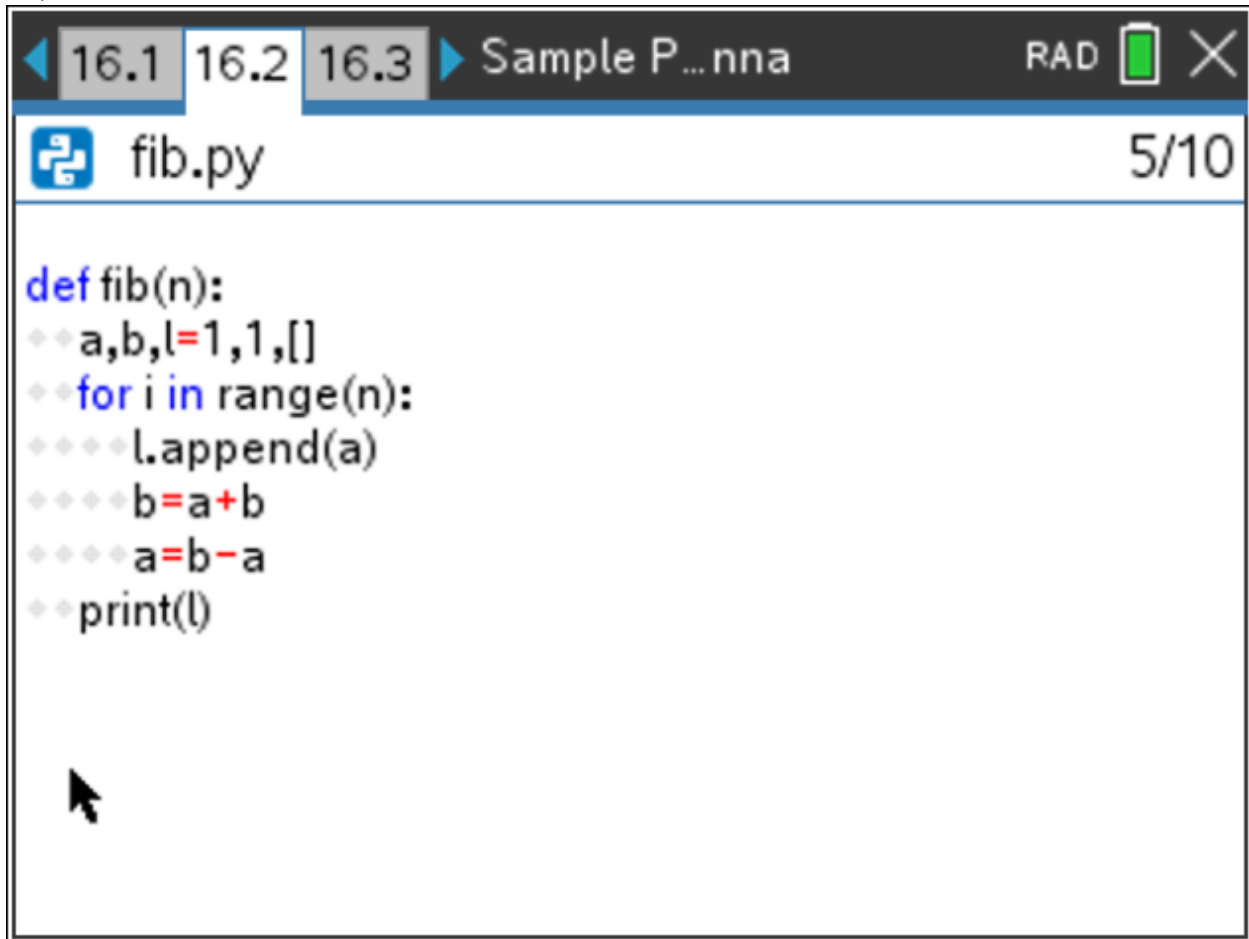
The screenshot shows a Jupyter Notebook window with a dark theme. The top bar contains navigation buttons (back, forward, search), a search bar with the text '*fizz buzz', and a 'RAD' button. The notebook title is 'segmentinsquare.py' and the page number is '3/12'. The code in the cell is as follows:

```
from math import *
from random import *
trials = 1000
segs=[]
for i in range(trials):
    x1=uniform(0,1)
    y1=uniform(0,1)
    x2=uniform(0,1)
    y2=uniform(0,1)
    d=sqrt((x1-x2)**2+(y1-y2)**2)
    segs.append(d)
print(sum(segs)/trials)
```

Fibonacci Sequence

Call the function `fib()` to create a sequence that is formed like the Fibonacci sequence. So...

`fib(10)` produces `[0,1,1,2,3,5,8,13,21,34,55]` and so on. It also displays the ratio of the last to numbers in the sequence. the function generates two numbers in each iteration. The `print()` function prints the requested number of numbers.

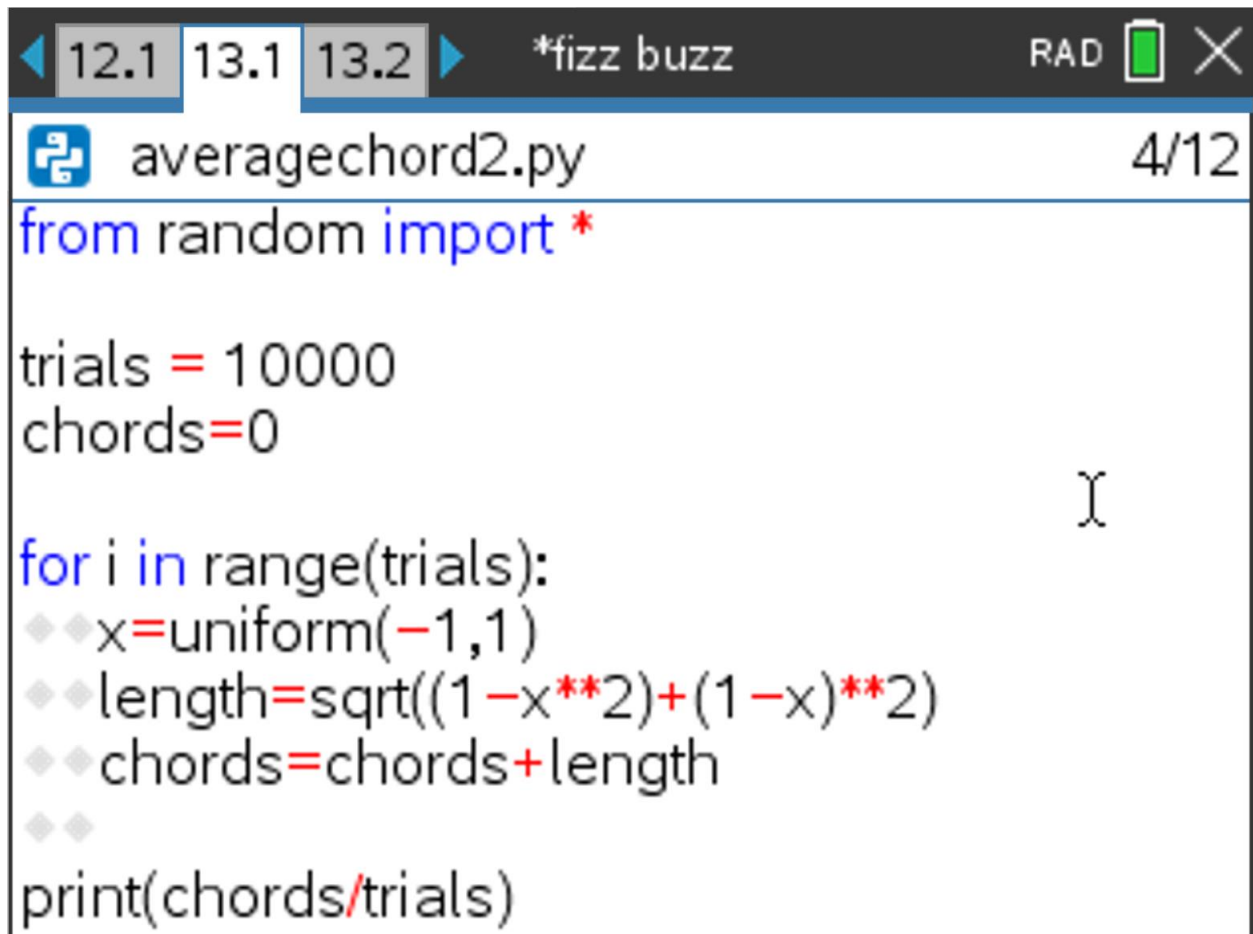




The screenshot shows a Jupyter Notebook window with a dark theme. The top bar contains navigation tabs labeled 16.1, 16.2, and 16.3, followed by a tab labeled 'Sample P... nna'. On the right side of the top bar, there is a 'RAD' label, a green battery icon, and a close button (X). Below the top bar, the notebook's title bar shows a file icon, the filename 'fib.py', and the page number '5/10'. The main area of the notebook displays a Python script for generating the Fibonacci sequence. The code is as follows:

```
def fib(n):  
    a,b,l=1,1,[]  
    for i in range(n):  
        l.append(a)  
        b=a+b  
        a=b-a  
    print(l)
```

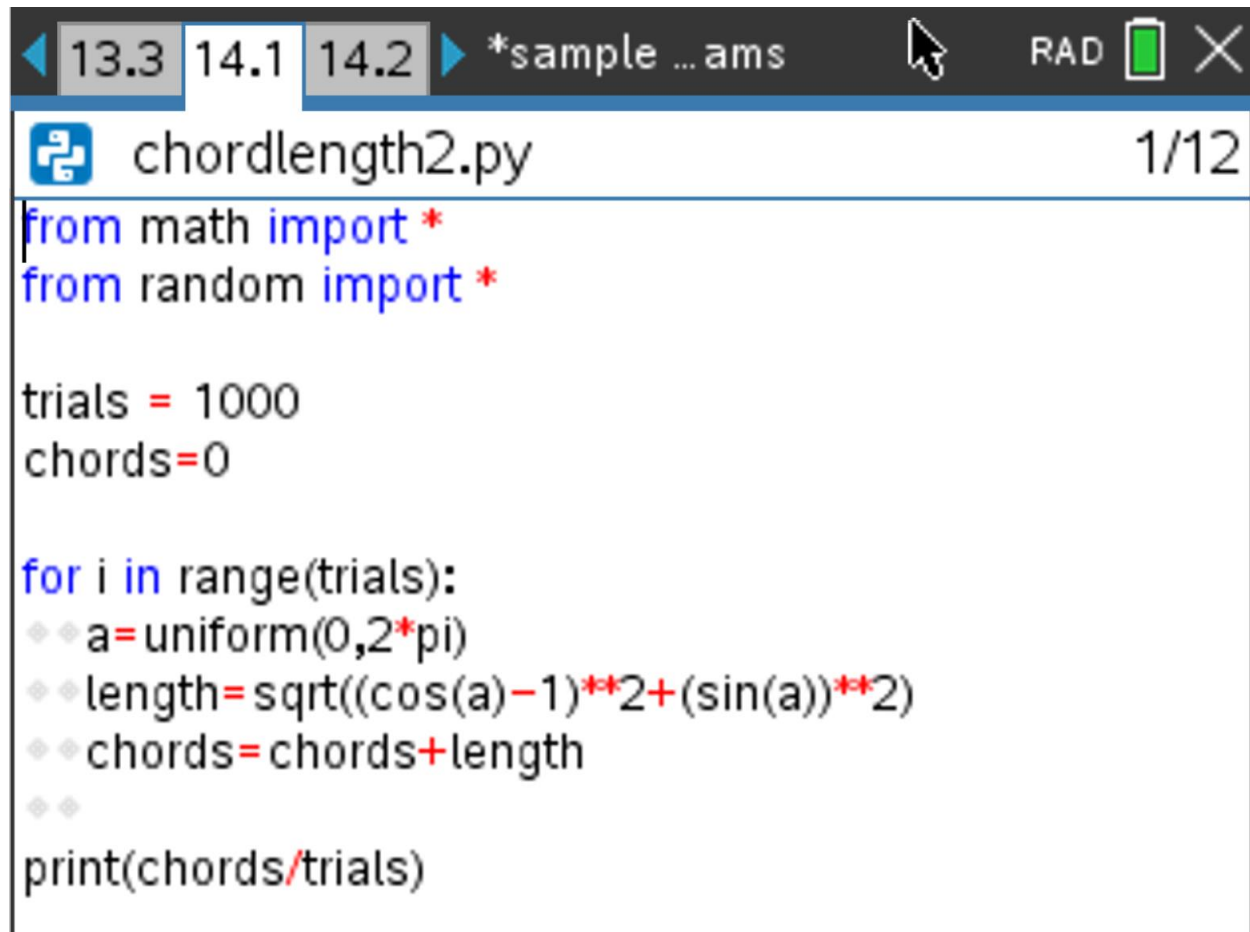
A mouse cursor is visible at the bottom left of the code area.

Average Length of a chord (version 1)



```
12.1 13.1 13.2 *fizz buzz RAD    
averagechord2.py 4/12  
from random import *  
  
trials = 10000  
chords=0  
  
for i in range(trials):  
    x=uniform(-1,1)  
    length=sqrt((1-x**2)+(1-x)**2)  
    chords=chords+length  
      
print(chords/trials)
```

Average Length of a Chord (version 2)



The screenshot shows a Jupyter Notebook window with a dark theme. The top bar contains navigation buttons (back, forward), a tab labeled '13.3 14.1 14.2', a file name '*sample ...ams', and status icons for 'RAD' and a battery level. The main area displays a Python script named 'chordlength2.py' at line 1/12. The script calculates the average length of a chord by sampling random points on a circle and averaging the distances between them.

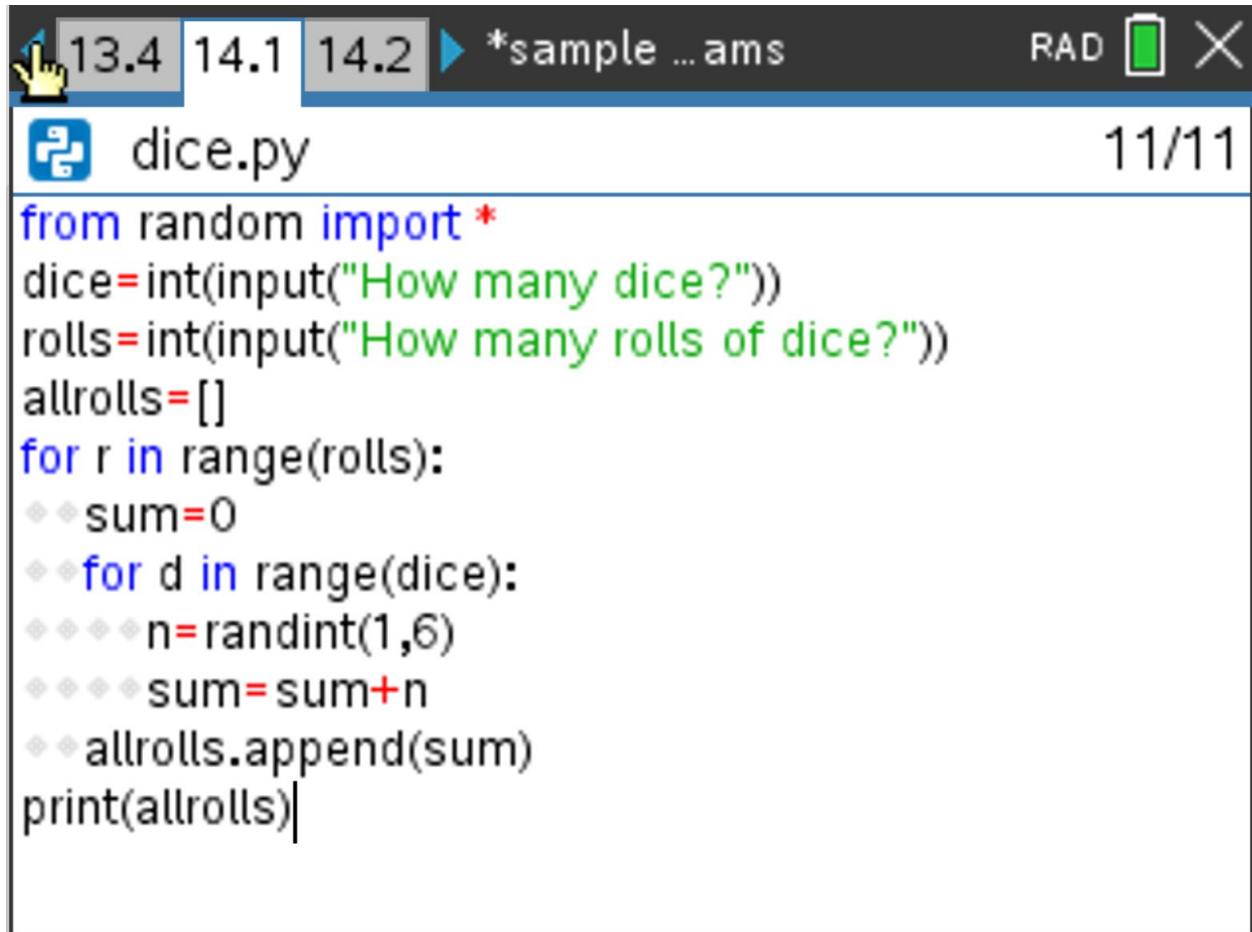
```
from math import *
from random import *

trials = 1000
chords=0

for i in range(trials):
    a=uniform(0,2*pi)
    length=sqrt((cos(a)-1)**2+(sin(a))**2)
    chords=chords+length
    ##
print(chords/trials)
```

Roll dice and record their sums

Run this program, choose the number of dice to roll, and choose how many rolls (trials) to perform. This will record the sums of all your rolls.



The screenshot shows a Jupyter Notebook interface. At the top, there are three tabs labeled 13.4, 14.1, and 14.2. The 14.1 tab is active, and it contains a file named 'dice.py'. The code in the file is as follows:

```
from random import *
dice=int(input("How many dice?"))
rolls=int(input("How many rolls of dice?"))
allrolls=[]
for r in range(rolls):
    sum=0
    for d in range(dice):
        n=randint(1,6)
        sum=sum+n
    allrolls.append(sum)
print(allrolls)
```

The code is color-coded: 'from' is blue, 'import' is blue, '*' is red, 'dice=' is black, 'int(input(' is green, 'How many dice?'))' is green, 'rolls=' is black, 'int(input(' is green, 'How many rolls of dice?'))' is green, 'allrolls=[]' is black, 'for' is blue, 'r in' is blue, 'range(rolls):' is black, 'sum=' is black, '0' is black, 'for' is blue, 'd in' is blue, 'range(dice):' is black, 'n=randint(1,6)' is black, 'sum=sum+n' is black, 'allrolls.append(sum)' is black, and 'print(allrolls)' is black. The cursor is at the end of the last line of code.

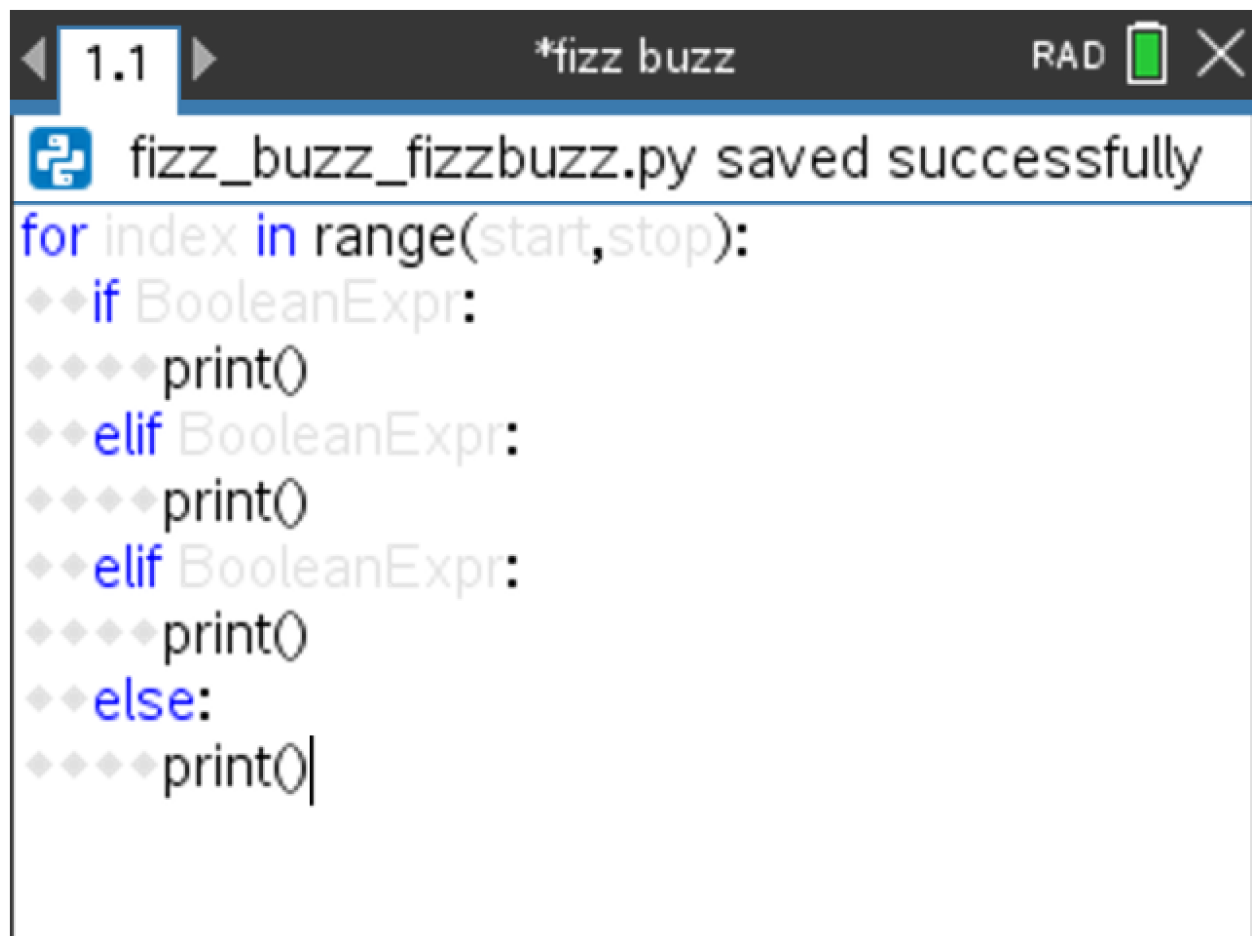
More Projects

Project: “Fizz Buzz FizzBuzz”

Write a program that prints the counting numbers but replaces multiples of 3 with the word “Fizz”, multiples of 5 with the word “Buzz” and multiples of both 3 and 5 with the word “FizzBuzz”.

Possible adaptations:

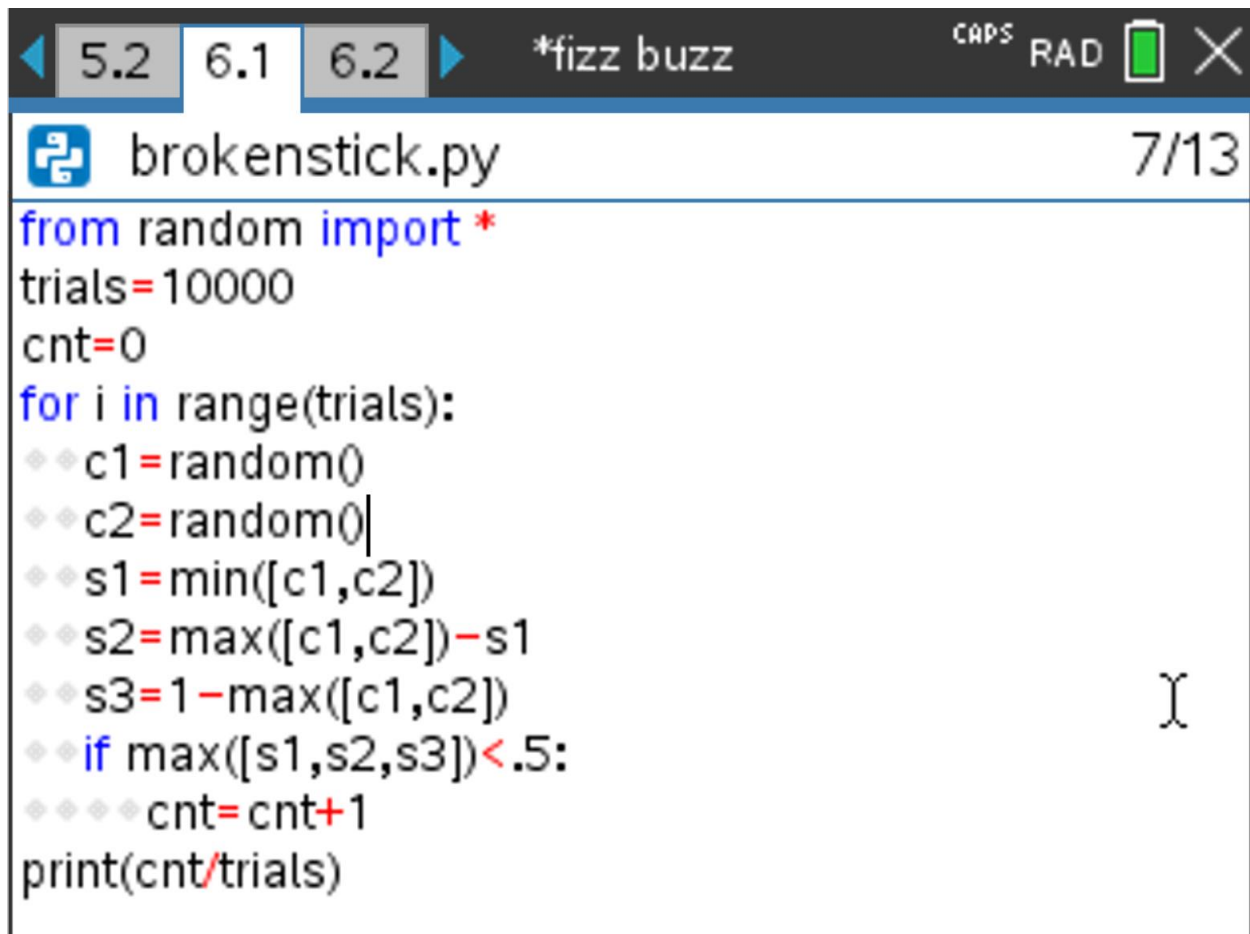
Input a lower and upper limit of numbers



```
1.1 *fizz buzz RAD [Battery Icon] [Close Icon]
fizz_buzz_fizzbuzz.py saved successfully
for index in range(start, stop):
    if BooleanExpr:
        print()
    elif BooleanExpr:
        print()
    elif BooleanExpr:
        print()
    else:
        print()
```

Broken Stick: Is it a Triangle?

A stick is cut in two random places - c_1 and c_2 - forming three pieces - s_1 , s_2 , and s_3 - that are used to form a triangle. Sometimes the pieces form a triangle, sometimes they don't. How likely is it the three pieces form a triangle?



The image shows a Jupyter Notebook window with a dark theme. The top bar contains navigation buttons (back, forward), a search bar with the text '*fizz buzz', and status indicators for 'CAPS', 'RAD', a battery icon, and a close button. The notebook title bar shows 'brokenstick.py' and '7/13'. The code cell contains the following Python code:

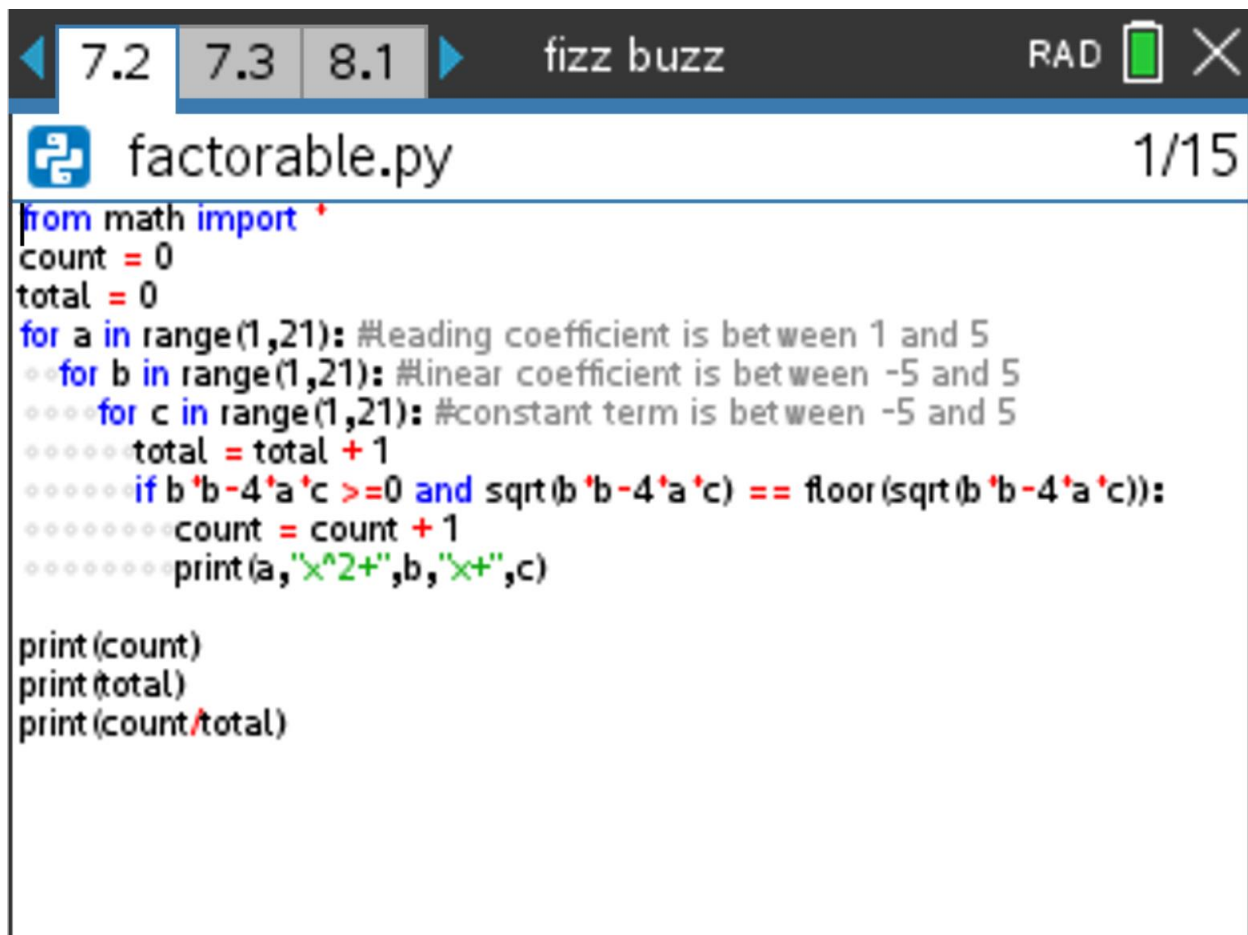
```
from random import *
trials=10000
cnt=0
for i in range(trials):
    c1=random()
    c2=random()
    s1=min([c1,c2])
    s2=max([c1,c2])-s1
    s3=1-max([c1,c2])
    if max([s1,s2,s3])<.5:
        cnt=cnt+1
print(cnt/trials)
```

The Game of Bagels

John Hanna, 11/28/2019

```
from random import *
print("Bagels")
def bagels():
    yn="y"
    while yn=="y":
        count=x=a=b=c=0 #initialize variables
        while a == b or b == c or a == c:
            a = randint(0,9)
            b = randint(0,9)
            c = randint(0,9)
        #endwhile
        ans=(100*a + 10*b + c)
        while x != ans:
            x = int(input("Guess: "))
            if x == 0:print(ans)
            count += 1
            h = int(x/100)
            t = int((x-100*h)/10)
            u = x-100*h-10*t
            str=""
            if h == a:
                str=str+"Fermi "
            if t == b:
                str=str+"Fermi "
            if u == c:
                str=str+"Fermi "
            if h == b:
                str=str+"Pico "
            if h == c:
                str=str+"Pico "
            if t == a:
                str=str+"Pico "
            if t == c:
                str=str+"Pico "
            if u == a:
                str=str+"Pico "
            if u == b:
                str=str+"Pico "
            if str=="":str="    Bagels"
            print(str)
        #endwhile
        print("YOU GOT IT IN", count, "guesses.")
        yn=input("Play again (y/n)?")
    #end of bagels()
#next line runs the program right away...
bagels()
```


Is it factorable? (probability that a quadratic is factorable)



The screenshot shows a Jupyter Notebook interface. The top bar has tabs for '7.2', '7.3', and '8.1', with '7.2' selected. The notebook title is 'fizz buzz'. On the right, there are icons for 'RAD', a battery level indicator, and a close button. The main area displays a Python script named 'factorable.py' (indicated by a file icon and the filename). The script is as follows:

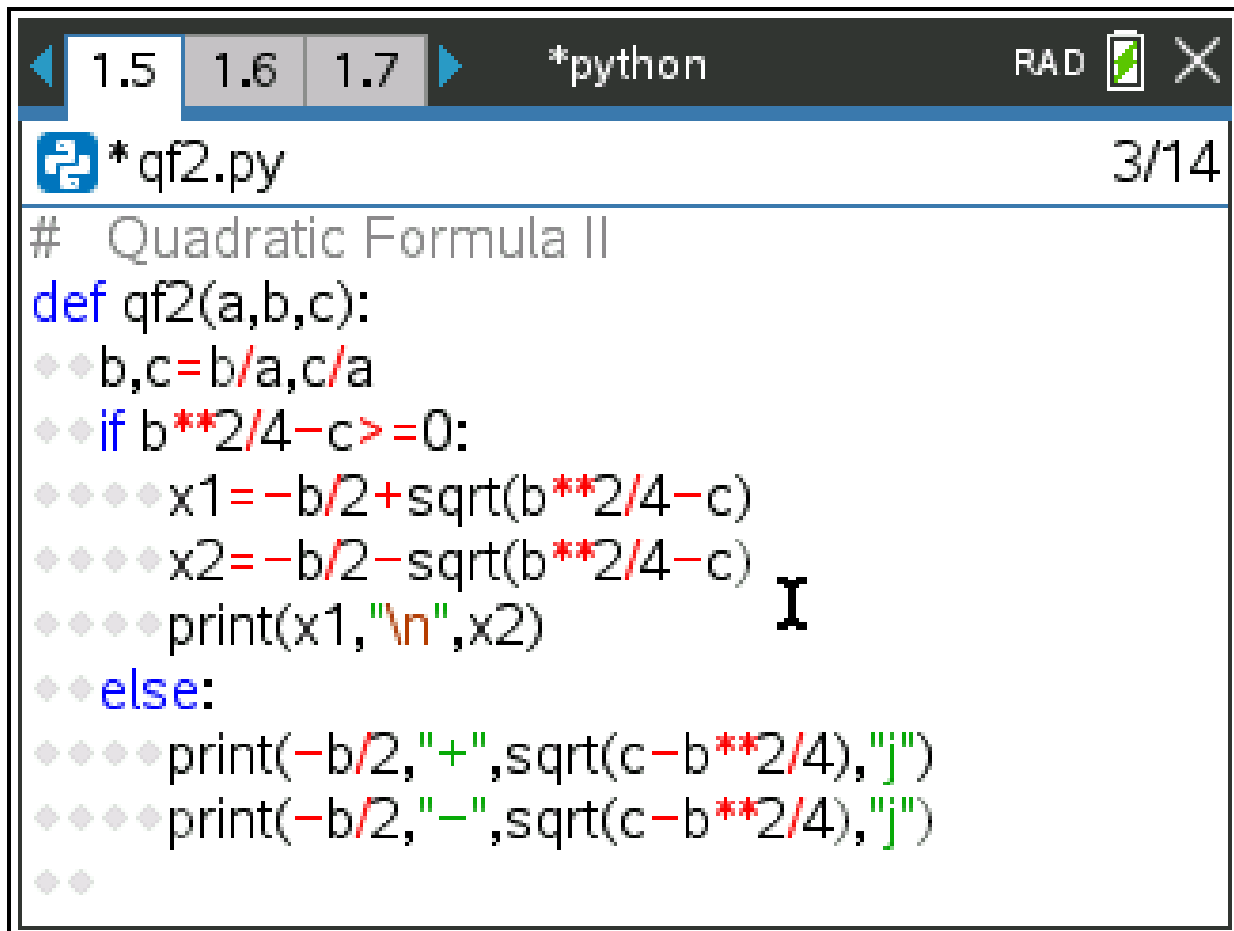
```
from math import *
count = 0
total = 0
for a in range(1,21): #leading coefficient is between 1 and 5
    for b in range(1,21): #linear coefficient is between -5 and 5
        for c in range(1,21): #constant term is between -5 and 5
            total = total + 1
            if b*b-4*a*c >=0 and sqrt(b*b-4*a*c) == floor(sqrt(b*b-4*a*c)):
                count = count + 1
            print(a,"x^2+",b,"x+",c)

print(count)
print(total)
print(count/total)
```

The script counts the number of factorable quadratics of the form $ax^2 + bx + c$ where a, b, c are integers between 1 and 20. It uses the discriminant $b^2 - 4ac$ to check if the quadratic has integer roots.

Related: Quadratic Formula II

Alternate Quadratic Formula (complex numbers?):



A screenshot of a Python IDE window. The title bar shows tabs for 1.5, 1.6, and 1.7, with the active tab labeled *python. On the right of the title bar are icons for RAD mode, a battery status icon, and a close button. The main editor area shows a file named *qf2.py with 3/14 lines of code. The code is a Python function qf2(a,b,c) that calculates the roots of a quadratic equation using the alternate quadratic formula for complex numbers. It includes comments and uses indentation for logic flow.

```
# Quadratic Formula II
def qf2(a,b,c):
    b,c=b/a,c/a
    if b**2/4-c>=0:
        x1=-b/2+sqrt(b**2/4-c)
        x2=-b/2-sqrt(b**2/4-c)
        print(x1,"\n",x2)
    else:
        print(-b/2,"+",sqrt(c-b**2/4),"j")
        print(-b/2,"-",sqrt(c-b**2/4),"j")
    
```

Reverse Raffle

```
from random import *
r = input("How many numbers?") #input will be a string
numbers=range(int(r)) #int(r) take input and makes it an integer
nn=len(numbers)
count = 0
print "Start", numbers
while len(numbers)>0:
    count = count + 1
    n = random.randint(0,nn-1)
    if n in numbers:
        numbers.remove(n)
    else:
        numbers.append(n)
    print count,n,sorted(numbers)
```

EXAMPLE

How many numbers? 5

Start [0, 1, 2, 3, 4]

iteration, number chosen, updated list.

1	2	[0, 1, 3, 4]
2	3	[0, 1, 4]
3	4	[0, 1]
4	4	[0, 1, 4]
5	2	[0, 1, 2, 4]
6	1	[0, 2, 4]
7	0	[2, 4]
8	1	[1, 2, 4]
9	1	[2, 4]
10	4	[2]
11	3	[2, 3]
12	2	[3]
13	2	[2, 3]
14	4	[2, 3, 4]
15	4	[2, 3]
16	3	[2]
17	2	[]

Project: Make Bricks

We want to make a row of bricks that is **goal** inches long. We have some small bricks (1 inch each) and some big bricks (5 inches each). Return True if it is possible to make the goal by choosing from the given numbers of bricks. This is a little harder than it looks and can be done without any loops.

Example function returns:

```
def make_bricks(small, big, goal):  
  
    make_bricks(3, 1, 8) → True  
    make_bricks(3, 1, 9) → False  
    make_bricks(3, 2, 10) → True
```

Why Is This Hard At All? #1

- You have 2 big bricks and 2 small bricks
- Can you make a row of 7 inches?
- Can you make a row of 8 inches?
- Notice: need small bricks to hit the length exactly

Why Is This Hard At All? #2

- Suppose you have 2 big bricks and 10 small bricks
- Can you make a row of 16 inches?
- Note: use lots of small bricks instead of a big brick