Lab 2 Overview: Symbolic Expressions, Graphs, and Equations

Last week, you learned how to use Python as a calculator, including assigning values to variables, creating symbolic variables, and substituting values into symbolic expressions. Today, we want to look back at our box example and learn how to rewrite expressions, solve equations (exact and approximate), and plot graphs of expressions.

EXAMPLE:

An open-top box is made by cutting out equal square corners of an 8.5 x 11 inch sheet of cardboard and folding up the flaps.

1) Write and expand a formula V(x) for the volume of the box as a function of the length x of the squares.

2) Find the volume when the squares are 2, 2 1/8, 2 2/8, 2 3/8, ..., 3 inch side lengths (every 1/8 inch from 2 to 3 inclusive).

3) Find the lengths required to get a volume of 50 cubic inches.

4) Plot a graph of V(x) in an appropriate practical domain.

Of course, nothing in Python can be done here without YOU doing the first step-writing the function V(x)! As you may recall from last week, our Volume function is V(x) = x *(8.5-2x)* (11 - 2x).

Once you have your function, you are ready to use Python. Recall that the symbolic package is a "library" of commands, so we will start all of our labs with the next set of commands which allow us to import all of the commands in sympy (i.e., "check out all the library commands")

In [1]:
```
from sympy import *
from sympy.plotting import (plot, plot_parametric)
```

1. Defining and Rewriting Symbolic Expressions

Now we are ready to define our symbolic variable x and start solving the problem. There are several commands that can be used to rewrite a symbolic expresssion, including **factor**, **expand**, and **simplify**. We'll show you what they all do this expression here; since #1 asks you to "expand" the expression, it is no surprise which command works best.

In [14]:
```
x=symbols('x')
# Recall this allows Python to treat the letter x as a variable.
V=x*(8.5-2*x)*(11-2*x)
print('Using simplify:',V.simplify()) # Remember to include explanatory tex
t in your print statements!
print('Using factor:',V.factor())
print('Using expand:',V.expand())
```

```
Using simplify: x*(2*x - 11)*(2*x - 8.5)
Using factor: 8.5*x*(0.235294117647059*x - 1.0)*(2*x - 11)
Using expand: 4*x**3 - 39.0*x**2 + 93.5*x
```

Two important things to notice in the above commands and output. First, as expected, the "expand" command expands our volume function, but recall from last week that the syntax is VERY different-and very common to Python! In most cases, when you want to perform a command on a variable, the correct Python syntax is

variable.comand

instead of "command(variable)". The second thing to notice is the Python output, specifically how exponents are used. Instead of printing 4 * x^3 (as done on a calculator, for example), Python used **for the exponent. This is because the ^ is a logical operator in Python. Inputs are the same: use x** 2 instead of x^2.

2. Substituting Into Symbolic Expressions

For question 2), we need to SUBSTITUTE the values into x. Recall that the **subs** command does the trick. Also recall from last week the idea of "list comprehension", where you can substitute ALL the values at once. We'll just type them all in here (noting that 2 1/8 = 2 + 1/8 and so on), but if you want, you can look up a command **arange**, which is part of the **numpy** (numerical python) package which makes it easier to enter them.

In [15]:
```
xvals=[2,2+1/8,2+2/8,2+3/8,2+4/8,2+5/8,2+6/8,2+7/8,3]
Volumes=[V.subs(x,i) for i in xvals]
# in words: create a list by taking the expression V and substituting for x
EVERY value in the list 'xvals'
print('Lengths of the squares (in inches):',xvals)
print('Volumes (in cubic inches):',Volumes)
```

```
Lengths of the squares (in inches): [2, 2.125, 2.25, 2.375, 2.5, 2.625, 2.7
5, 2.875, 3]
Volumes (in cubic inches): [63.0000000000000, 60.9609375000000, 58.50000000
00000, 55.6640625000000, 52.5000000000000, 49.0546875000000, 45.37500000000
00, 41.5078125000000, 37.5000000000000]
```

3. Solving Equations Symbolically (Exact)

For question 3) we need to solve V(x) = 50. There is a type "Equation" in Python, but it is generally easiest to move everything to one side and use the expression, in this case, solve V - 50 = 0. It should come as no surprise that we use the **solve** command. Notice that we don't have to use the *Variable.Command* syntax.

In [16]:
```
Vsoln=solve(V-50,x)
print('The values of x which make V=50 are',Vsoln)
```

```
The values of x which make V=50 are [0.753000478517084 + 0.e-22*I, 2.591605
67813342 - 0.e-22*I, 6.4053938433495 - 0.e-20*I]
```

Notice that Python gives what appear to be complex solutions. Looking carefully at our output (ALWAYS a good practice!), we see that the solutions end up with an infinitessimal imaginary residue. So we can ignore that. Rather than having to retype the long decimals, we can ask Python to return just the real part of the solutions. A quick check on help documentation shows that the command **re** does that. However, Python returned a LIST of solutions (**NOTE**: Python does this even when there is only ONE solution!), so we have to be careful about how we do it:

In [18]:
```
print(re(Vsoln))
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-18-d4f1e03e42e7> in <module>
----> 1 print(re(Vsoln))

~\Anaconda3\lib\site-packages\sympy\core\function.py in __new__(cls, *args,
**options)
    456
    457         evaluate = options.get('evaluate', global_evaluate[0])
--> 458         result = super(Function, cls).__new__(cls, *args, **options
)
    459         if evaluate and isinstance(result, cls) and result.args:
    460             pr2 = min(cls._should_evalf(a) for a in result.args)

~\Anaconda3\lib\site-packages\sympy\core\function.py in __new__(cls, *args,
**options)
    275
    276         if evaluate:
--> 277             evaluated = cls.eval(*args)
    278             if evaluated is not None:
    279                 return evaluated

~\Anaconda3\lib\site-packages\sympy\functions\elementary\complexes.py in ev
al(cls, arg)
     56         elif arg is S.ComplexInfinity:
     57             return S.NaN
---> 58         elif arg.is_real:
     59             return arg
     60         elif arg.is_imaginary or (S.ImaginaryUnit*arg).is_real:

AttributeError: 'list' object has no attribute 'is_real'
```

Notice a long, complicated looking error occurs here. The last line, however is the key: basically, Python is saying it cannot perform this function on a list. This is true of most symbolic functions. But we can find get the real part of each solution using list comprehension as we did in #2.

In [20]:
```
Vreal=[re(i) for i in Vsoln]
print('The solutions are when x=',Vreal)
```

```
The solutions are when x= [0.753000478517084, 2.59160567813342, 6.405393843
34950]
```

HOWEVER, if we think about our answer (and just because you are using a computer to do things doesn't mean you can stop thinking!), we should see that the last answer is nonesense. We can't cut 6 1/2 inches from each corner of an 8 1/2 x 11 sheet of cardboard! So we only want the first two solutions.

IMPORTANT!!! When counting the elements in a list, Python starts counting at ZERO, not one! So if we want the first two solutions, they are the 0th and 1st elements of the list. Notice the use of square brackets to refer to a specific element in the list.

In [22]:
```
print('The practical solutions are when x=',Vreal[0],'and',Vreal[1],'inches.')
```

```
The practical solutions are when x= 0.753000478517084 and 2.59160567813342
inches.
```

Perhaps an easier way to obtain the real, practical solutions (and for many equations, the ONLY way) is to actually proceed with #4 and use the graph to solve the equation numerically.

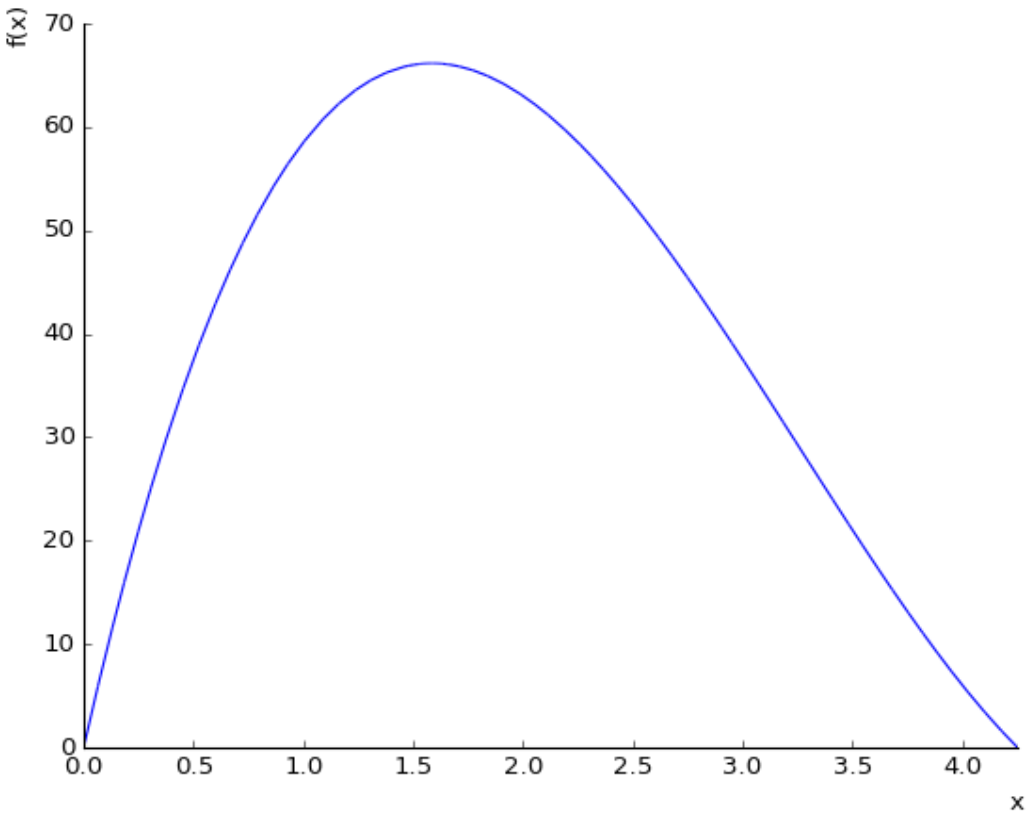
4. Plotting Symbolic Expressions

You will learn ways to numerically plot expressions in ENGR 102; we will focus on symbolic plotting here for now. In Jupyter, before each graph, you need to enter the following command:

In [12]:
```
matplotlib notebook
```

This allows Python to produce the graph in the Jupyter notebook (and not included on a previous graph). For a practical domain, we notice each of the terms we multiplied in our Volume must be positive. So 0<=x<=4.25.

In [5]:
```
plot(V,(x,0,4.25))
```



Out[5]:
```
<sympy.plotting.plot.Plot at 0x7f082d9645c0>
```

3. (Ctd) Solving Equations Numerically (Approximate)

From the graph, y=V(x)=50 when x is between 0.5 and 1.0 and also when x is about 2.5 (Closer to 2 5/8 if you observed your output in #2). So we can now use Python's **nsolve** command (numerically solve) by also including a starting "guess" near the solution

In [13]:
```
x1=nsolve(V-50,0.5)
x2=nsolve(V-50,2.5)
print('V=50 when x=',x1,x2,'inches.')
```

```
V=50 when x= 0.753000478517084 2.59160567813342 inches.
```

In [ ]:
```

```