tutorial014-Numpy-CreatingPythongFunctions

November 1, 2022

0.1 Reuse useful code by creating functions

0.1.1 Learning goals

- Understand what is a function in programming
- Understand the syntax used by Python to define a function
- Practice with writing a function
  - Write an example function to make data
  - Write an example function to make a plot

0.1.2 Recycling code

Imagine if you were to be taking a data science class and you were often asked to create:

- Datasets of different means and standard deviations, say using code like: `data = mu + sd*np.random.randn(siz,1)`
- Plots (like histograms) of the datasets. For example using `sns.kdeplot`.

You might be tempted to memorize, or even write down on a notepad, the specific code that seems to be used most often. Looking up the code snippet from a piece of paper it would be probably fine at least in a few cases.

Imagine now becoming a data scientist. Imagine handling dozens of code and analysis requests from multiple managers and clients. Many requests will likely require reusing very similar blocks of code. In theory, also in this case you *might be tempted* to use the same methid and copy and paste the same code from the piece of paper to apply to edit it and then apply it to each new request.

There are better options for you. Given that your work day will likely end only after all requests have been attended to, copying and pasting might be "suboptimal," to say the least.

0.1.3 Functions

Functions are code methods that allow to encapsulate useful code in a reusable form. Methods exist to speed up your data science work, reduce mustakes and, eliminate extra typing needed when reusing code. These methods are called **functions**.

Functions, a.k.a. subroutines are sequences of code packaged as units. The units can be called at different locations in code to quickly and efficiently perform the oprations implemented in the packaged code. Functions are used to address situations just like the ones described above, where a section of code is used over and over again. Functions accept inputs and return outputs. Functions, contain useful code that performs operations that are likely to be used multiple times.

Functions can help make code shorter, more nimble, and easier to read.

0.1.4 Flipping a coing to choose lunch

A coin flip operation is a good, easy, example to describe how functions work in python.

Imagine wanting to make a function that works just like the flip of a coin. For fun, imagine wanting to use the function to pick your lunch *Tacos!* or *Pizza!* being your options. Let's look at our first example of a function.

The code below is an example of how we can do something like that, simulate coinflips. To implement the code a function was defined using Python's syntax. In Python, functions are defined with a function name and the special word `def` in front of it.

[1]:
```python
def coinflip() :
    import numpy as np
    flip = np.random.randint(2) # return a random 0 or 1
    if flip == 0 :
        result = 'Pizza!'
    else :
        result = 'Tacos!'
    return result
```

[2]:
```python
# Let's test this function.
# Run this cell multiple times
coinflip()
```

[2]:
```
'Pizza!'
```

You should be returned either Pizza or Tacos at every coinflip call!

0.1.5 Anatomy of a function

Let's analyze the function above.

- def: The function begins with the word def.

```
def
```

- The function name: def is followed by the function's name. The name is chosen by the programmer to reflect what the function does.

```
def coinflip
```

- The parenthesis: The function name is followed by a pair of parenthesis `()`.

```
def coinflip()
```

- The column sign (`:`): The parenthesis are followed by a column sign.

```
def coinflip() :
```

- The code lines: Indented within the function definition are the "actual code" lines that make up the function. When a function runs, the computer runs these lines of code from top to bottom. Below the specific lines from the function at stake.

`import numpy as np    flip = np.random.randint(2) # return a random 0 or 1    if flip == 0 :       result = 'Tails!'    else :       result = 'Heads!'    return result`

Importantly, all the code that belongs inside the function must be indented so as to align with the first letter of the function name. In other words, the following is incorrect:

[3]:
```
def coinflip() :
    import numpy as np
    flip = np.random.randint(2) # return a random 0 or 1
    if flip == 0 :
        result = 'Tails!'
    else :
        result = 'Heads!'
    return result
```

Just like for loops, *indentation is critical in Python*!

Two additional bits of code are important to dissect from the code above. The second line of code `import numpy as np` in the block shows that imports must be done inside the function definition. This is because the function does not share the whole software environment and variables outside. So variables and operations inside a function will need their own definitions and imports.

In the case of our coinflip we import `numpy`.

`def coinflip() : import numpy as np`

Complete the following exercise.

- Let's test how the code inside the function communicates to the python environment outside of the function. To do so, use the cell below to define a numpy array `d` containing the the list of numbers `[2, 3, 4, 5, 6]`. Note that we are asking to do so without importing numpy again, but assuming that the `import` operation written inside the function above is all you need also to import outside the function.

[9]:
```
def d() :
    d = array[2,3,4,5,6]
    return(d)
```

- Where you able to define the numpy array? Why not? Use this cell to explain what is happening here, why are you suddenly not able to define a numpy array? What should you do before being able to define one?

  -Nope! the array is outside the function.

The variables and data generated by the function will need to be returned out to the outside environment and for that the special term `return` is used.

`flip = np.random.randint(2) # return a random 0 or 1 if flip == 0 :     result = 'Tails!' else :     result = 'Heads!' return result`

if that term is omitted nothing the values computed insdiethe function are not returned outside of the function. (Well, unfortunately it is more complicated than that but we will talk about the full story in a different tutorial.)

But let's test it. if we run a modified version of `coinflip` without the return line what happens?

[10]:
```
def coinflip() :
    import numpy as np
    flip = np.random.randint(2) # return a random 0 or 1
    if flip == 0 :
        result = 'Pizza!'
    else :
        result = 'Tacos!'
```

What is the result? (Hint: There should be no return, no free lunch. Try.) The `coinflip` values aren't returned outside of the function.

Complete the following exercise.

- Advanced question: Can you think a way to bypass (avoid) returing `result` but still showing the option selected for lunch (*Tacos* or *Pizza*)?

Maybe using `print()` could still return the options.

[20]:
```
def coinflip() :
    import numpy as np
    flip = np.random.randint(2) # return a random 0 or 1
    if flip == 0 :
        result = 'Pizza!'
    else :
        result = 'Tacos!'
    print(result)

coinflip()
```

```
Tacos!
```

0.1.6 A more advanced version of a function: A bread maker

So far we have been discussing a very simple case for a function. One that returns a single value to the outside world. The coin flip is a simple and limited case.

Often times functions need stuff to work. Just like a breadmaker: add flour, yeast, water, and salt to get some bread out! A function can act like bread maker, get raw inputs and return much needed outputs.

Let's go back to look at one of code operations we have encountered multiple times in previous tutorials.

`data = mu + sd*np.random.randn(siz,1)`

Several times, we have generated data by using a random generator. Often times we have used a generator that returns normally distributed data.

Imagine wanting to use what you just learned about functions to write a new function that can generate distributions of normally distributed data for you with a certain mean, standard deviation and size.

To do so, we would need a function that can return `data`. We could do that by reusing what just learned with `coinflip()`:

```
[12]: def my_normal_data():
          import numpy as np
          data = 5 + 2*np.random.randn(10,1)
          return data
```

We can take a look at the returned variable `data`:

```
[13]: d = my_normal_data()
```

```
[14]: import numpy as np

      # What is the mean value?
      np.mean(d)
```

```
[14]: 5.125105214616922
```

```
[15]: # What about the standard deviation?
      np.std(d)
```

```
[15]: 1.956385535170588
```

```
[16]: # How many data points did we get?
      np.size(d)
```

```
[16]: 10
```

If the function above is run multiple times, the results will aways return: - different numbers, - a mean around `5`, - a standard deviation around `2` and - a numerosity (the size) of the sample of `10`.

Complete the following exercise.

- Use the cell below to write a function called `mybrain` that returns a variable called `butterflies` that contains 15 numbers with mean -5 and standard deviation of 0.5.

```
[23]: def mybrain() :
          import numpy as np
          butterflies = -5 + 0.5*np.random.randn(15,1)
          return(butterflies)
```

Functions can return data by variable assigment Note that we have already encountered a difference from `coinflip()`.

The function `my_normal_data()` was called and the value returned by the function was assigned to a variable called `d`.

In the early example, `coinflip()` did not return usable data. All we wanted from the function was to visualize the value of a varible internal to the function.

In the case right above here, we wanted to actual data to compute the `mean`, the `std` and measure the `size`. To do so, we had to use additional functionality of python; we assigned the returning variable `data`, that existed internal to the function to the variable `d` which existed outside of the function.

It is `d` that we can use as data and it is `d` that we can study. So a new piece of learning here: - Python functions can return internal variables but the variables must be assigned to output variables to be used outside of the function.

The bread maker This is all great.

We have now learned how to create an advanced function that returns data. Yet, this is not quite a breadmaker yet!

Wouldn't it be terrific if the function could receive as inputs the size, the mean and the standard deviation of the data sample? That would allow a user to specify the center and spread of the distribution of data as well as the size of the data set.

That would look pretty much like a bread maker: We add ingredients and return freshly baked awesome bread.

Functions can accept variables as inputs and use those variables to perform operations Python function can take inputs just like they can return outputs. *Input variables* can be added at the function definition and the function can then use the values stored inside the variables for internal computations.

Below an example.

[24]:
```
def my_awesome_data(mu,sd,siz):
    import numpy as np
    data = mu + sd*np.random.randn(siz,1);
    return data
```

[33]:
```
d = my_awesome_data(0,2,20)
```

[34]:
```
# What is the mean value?
np.mean(d)
```

[34]:
```
0.42941332178535924
```

[35]:
```
# What about the standard deviation?
np.std(d)
```

```
[35]: 1.901617784662691
```

```
[36]: # How many data points did we get?
      np.size(d)
```

```
[36]: 20
```

What happens if you change the inout values say from `(0,2,20)` to `(10,4,200)`? Give it a try:

```
[37]: d = my_awesome_data(10,4,200)
```

Complete the following exercise.

- What happens if you omit the inputs? Use the cell below to first try to run the function omitting the output and then use the following cell to explain what is going on using your own words (Hint: What does the error say?)

```
[41]: def my_awesome_data(mu,sd,siz):
          import numpy as np
          data = mu + sd*np.random.randn(siz,1);
          return data
```

It does not work because there are no specified values for the input variables.

```
[42]: # add example of non positional inout assignment
      d = my_awesome_data(sd=4,siz=200,mu=10)
      np.mean(d)

      # add example of default values asigment to the function
      def my_awesome_data(mu=10,sd=2,siz=100):
          import numpy as np
          data = mu + sd*np.random.randn(siz,1);
          return data
```

This is an awesome new function that doesn't just return data, but it can return the data you need. You can pick the flower (the mean), the salt (the standard deviation) and the yeast (the size of the dataset) and the function will make bread for you, or better than that, the function will give you the data you want.

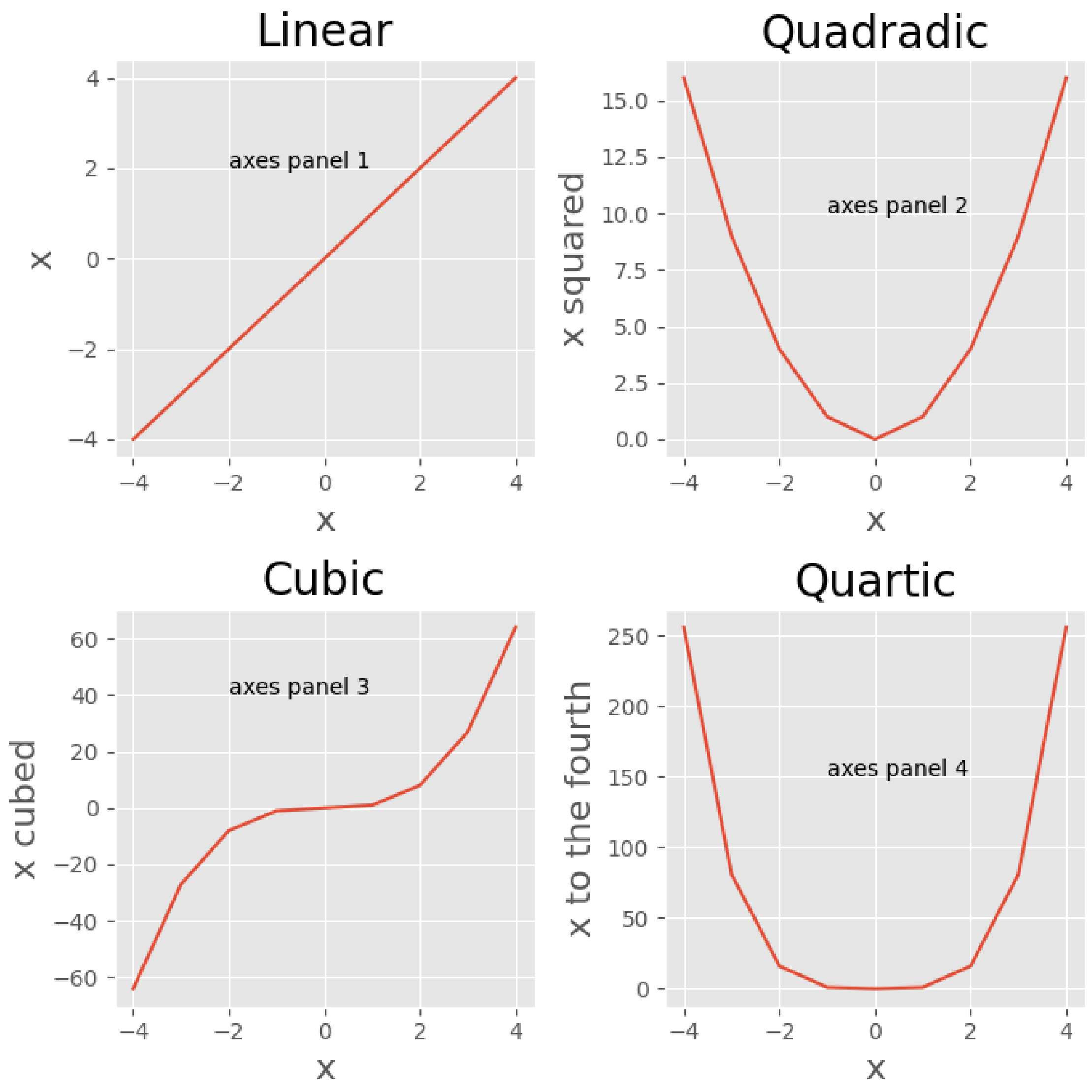
The function can be reused, copied, pasted and once defined in a jupyter notebook it can be used *ad infinitum* (again and again).

0.1.7 Combining functions to simplify and reuse plotting code

In a past tutorial, we have learned how to use `matplotlib.pyplot` to make pretty plots. The code of one of the past examples is reproduced below. It makes four pretty plots of a Linear, a Quandratic, a Cubic and a Quartic function.

```
[43]: # make sure we have our tools
      import numpy as np
```

```python
import matplotlib.pyplot as plt

# compute our polynomials
my_x = np.array([-4, -3, -2, -1, 0, 1, 2, 3, 4])
y_lin = my_x
y_quad = my_x**2  # ** is Python for exponentiation
y_cube = my_x**3
y_quart = my_x**4

# set our style
plt.style.use('ggplot')
axisLabelSize = 16
titleSize = 20

# make a big square figure
plt.figure(figsize=(7, 7))

# Now make each of the plots!

# first is top left
plt.subplot(221)
plt.plot(my_x, y_lin, label = 'linear');
plt.xlabel('x', fontsize = axisLabelSize);
plt.ylabel('x', fontsize = axisLabelSize);
plt.title('Linear', fontsize = titleSize);
plt.text(-2, 2, 'axes panel 1')

# then top right
plt.subplot(222)
plt.plot(my_x, y_quad, label = 'quadratic');
plt.xlabel('x', fontsize = axisLabelSize);
plt.ylabel('x squared', fontsize = axisLabelSize);
plt.title('Quadradic', fontsize = titleSize,);
plt.text(-1, 10, 'axes panel 2')

# third is bottom left
plt.subplot(223)
plt.plot(my_x, y_cube, label = 'cubic');
plt.xlabel('x', fontsize = axisLabelSize);
plt.ylabel('x cubed', fontsize = axisLabelSize);
plt.title('Cubic', fontsize = titleSize,);
plt.text(-2, 40, 'axes panel 3')

# then bottom right
plt.subplot(224)
plt.plot(my_x, y_quart, label = 'quartic');
plt.xlabel('x', fontsize = axisLabelSize);
plt.ylabel('x to the fourth', fontsize = axisLabelSize);
plt.title('Quartic', fontsize = titleSize,);
plt.text(-1, 150, 'axes panel 4')

# finally, make everything automagically fit
plt.tight_layout()
```



We have to admit that the plotting code is long and ugly.

We are going to use a function to separate the plotting code from the data generation code so to be able to call this whole block of code multiple times.

First of all we have the data generation block of code.

[44]:
```
# This is the data generation code
import numpy as np
import matplotlib.pyplot as plt

# compute our polynomials
my_x = np.array([-4, -3, -2, -1, 0, 1, 2, 3, 4])
y_lin = my_x
y_quad = my_x**2  # ** is Python for exponentiation
y_cube = my_x**3
y_quart = my_x**4
```

After that, we have the plotting code. We are not going to stick that code into a function.

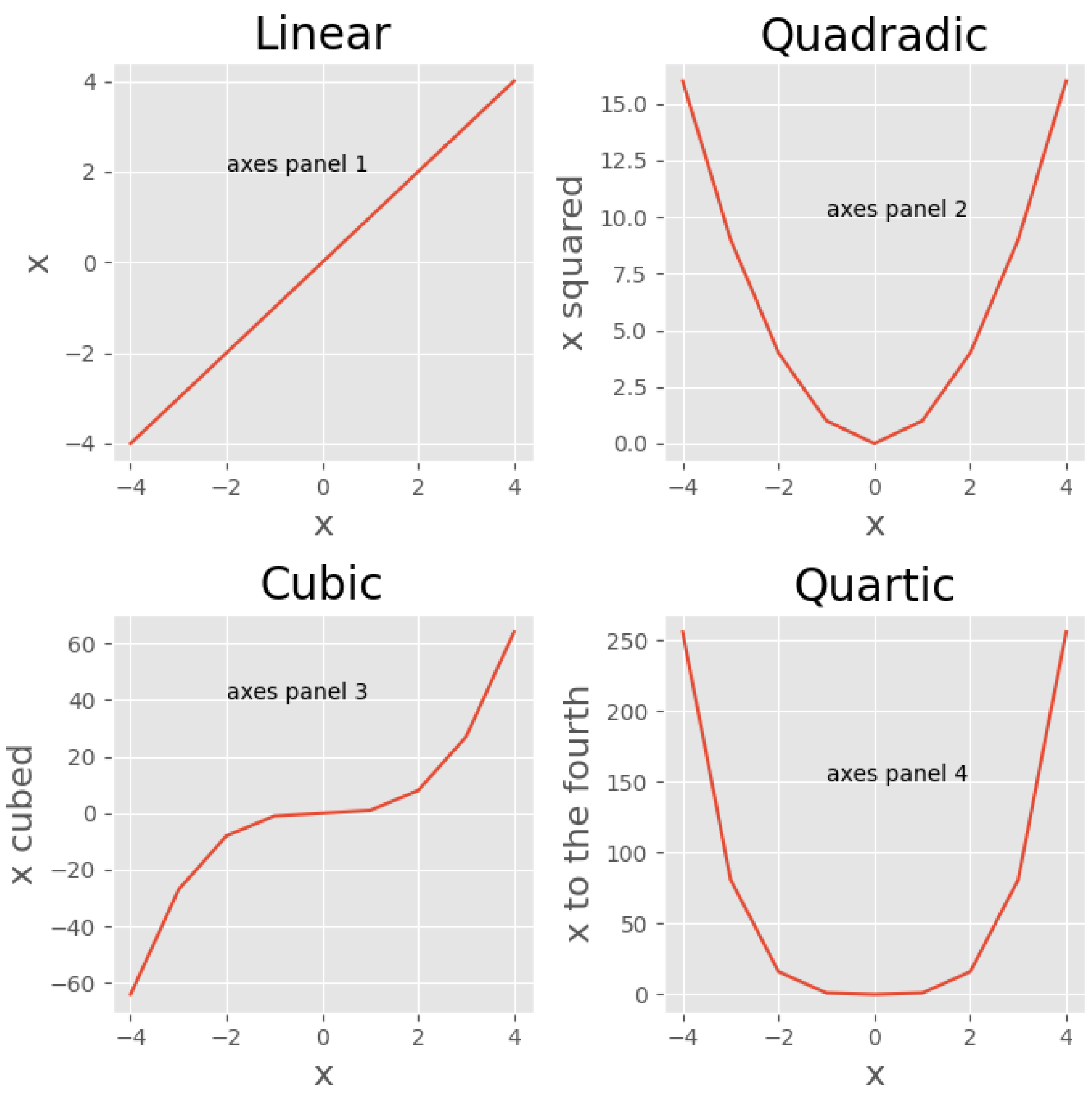
The function will need as inputs the data we generated outside: - my_x - y_lin - y_quad - y_cube - y_quart

The rest of the code will be just like above. The new function will take as inputs the data (5 vectors of the same size) and make plots:

[45]:
```
def my_spiffy_plotting_func(my_x, y_lin, y_quad, y_cube, y_quart):
    import numpy as np
    import matplotlib.pyplot as plt # This is the function that will plot the␣
 ↪data

    # set our style
    plt.style.use('ggplot')
    axisLabelSize = 16
    titleSize = 20

    # make a big square figure
    plt.figure(figsize=(7, 7))

    # Now make each of the plots!

    # first is top left
    plt.subplot(221)
    plt.plot(my_x, y_lin, label = 'linear');
    plt.xlabel('x', fontsize = axisLabelSize);
    plt.ylabel('x', fontsize = axisLabelSize);
    plt.title('Linear', fontsize = titleSize);
    plt.text(-2, 2, 'axes panel 1')

    # then top right
    plt.subplot(222)
    plt.plot(my_x, y_quad, label = 'quadratic');
    plt.xlabel('x', fontsize = axisLabelSize);
    plt.ylabel('x squared', fontsize = axisLabelSize);
    plt.title('Quadradic', fontsize = titleSize,);
    plt.text(-1, 10, 'axes panel 2')

    # third is bottom left
    plt.subplot(223)
    plt.plot(my_x, y_cube, label = 'cubic');
    plt.xlabel('x', fontsize = axisLabelSize);
    plt.ylabel('x cubed', fontsize = axisLabelSize);
    plt.title('Cubic', fontsize = titleSize,);
    plt.text(-2, 40, 'axes panel 3')

    # then bottom right
    plt.subplot(224)
    plt.plot(my_x, y_quart, label = 'quartic');
    plt.xlabel('x', fontsize = axisLabelSize);
    plt.ylabel('x to the fourth', fontsize = axisLabelSize);
    plt.title('Quartic', fontsize = titleSize,);
    plt.text(-1, 150, 'axes panel 4')

    # finally, make everything automagically fit
    plt.tight_layout()
```

Let's next launch the the function passing the data!

```
[46]: my_spiffy_plotting_func(my_x, y_lin, y_quad, y_cube, y_quart)
```



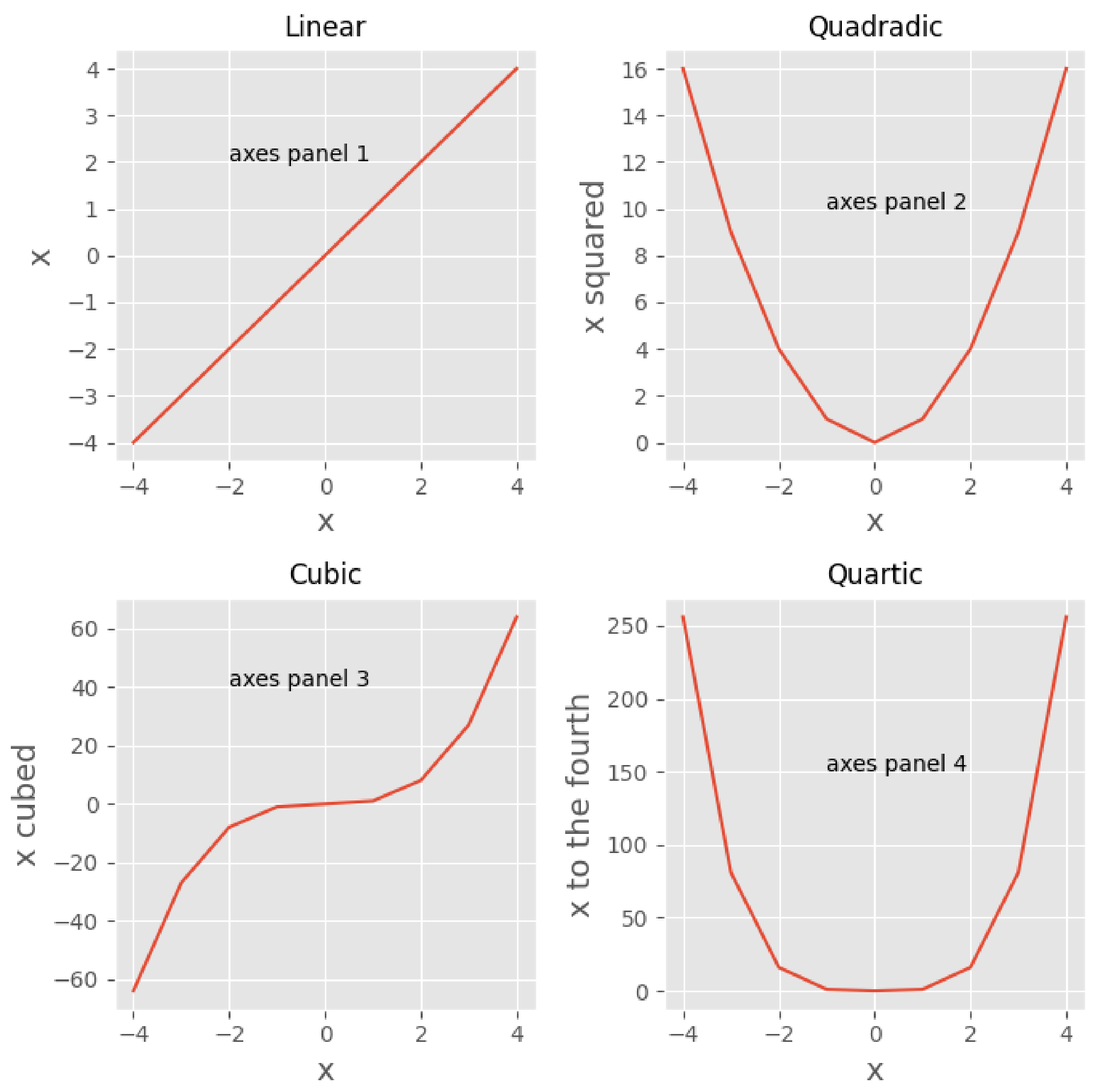
Isn’t that nice? We have a one line of code now that can make that pretty plot of ours.

What if we wanted to change the font size later one? We could add additional parameters to allow us to variate the font size everytime we call the function. Let’s do that. We will redefine the function adding the font size as input.

[47]:
```
def my_spiffy_plotting_func(my_x, y_lin, y_quad, y_cube, y_quart,␣
  ↪axisLabelSize=20, titleSize=16,):
# This is the function that will plot the data
    # set our style
    plt.style.use('ggplot')

    # make a big square figure
    plt.figure(figsize=(7, 7))
    # Now make each of the plots!

    # first is top left
    plt.subplot(221)
    plt.plot(my_x, y_lin, label = 'linear');
    plt.xlabel('x', fontsize = axisLabelSize);
    plt.ylabel('x', fontsize = axisLabelSize);
    plt.title('Linear', fontsize = titleSize);
    plt.text(-2, 2, 'axes panel 1')

    # then top right
    plt.subplot(222)
    plt.plot(my_x, y_quad, label = 'quadratic');
    plt.xlabel('x', fontsize = axisLabelSize);
    plt.ylabel('x squared', fontsize = axisLabelSize);
    plt.title('Quadradic', fontsize = titleSize,);
    plt.text(-1, 10, 'axes panel 2')

    # third is bottom left
    plt.subplot(223)
    plt.plot(my_x, y_cube, label = 'cubic');
    plt.xlabel('x', fontsize = axisLabelSize);
    plt.ylabel('x cubed', fontsize = axisLabelSize);
    plt.title('Cubic', fontsize = titleSize,);
    plt.text(-2, 40, 'axes panel 3')

    # then bottom right
    plt.subplot(224)
    plt.plot(my_x, y_quart, label = 'quartic');
    plt.xlabel('x', fontsize = axisLabelSize);
    plt.ylabel('x to the fourth', fontsize = axisLabelSize);
    plt.title('Quartic', fontsize = titleSize,);
    plt.text(-1, 150, 'axes panel 4')

    # finally, make everything automagically fit
    plt.tight_layout()
```

We can now run this function. We will pass slightly different font size (14 and 12, instead of 16 and 20).

[48]:
```
my_spiffy_plotting_func(my_x, y_lin, y_quad, y_cube, y_quart, 14, 12)
```



Sweet.

In this tutorial we have learned the basic syntax of functions in python.

The definition, the inputs, the ouputs. We have also practiced with defining two types of functions, one that can be used to generate data. The other, that can be used to plot data.

There many applications to functions and indeed the majority of what we have been using so far in Python is defined as a function! There are also other apsects of functions and how they work that we have not covered above.

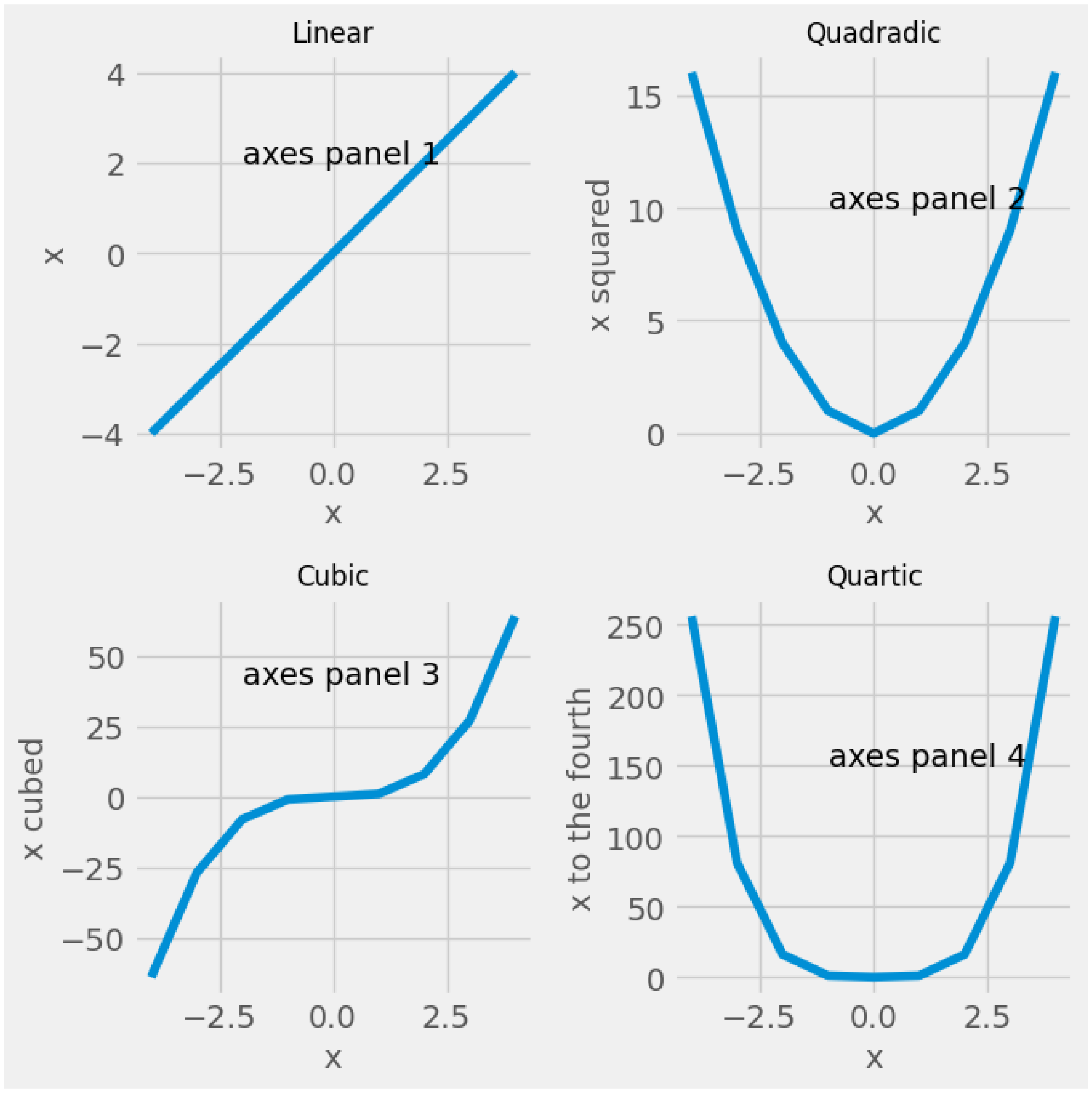
Complete the following exercise.

- If we wanted to be able to change the style of the plots above, say we would like to pick 'fivethirtyeight' or other styles, how would you modify the definition of `my_spiffy_plotting_func()` so to allow changing the style, every time the function is called? Use the cell below to answer the question.

```
[54]: def my_spiffy_plotting_func(my_x, y_lin, y_quad, y_cube, y_quart,␣
       ↪axisLabelSize=20, titleSize=16, plot_style='ggplot'):
      # This is the function that will plot the data
          # set our style
          plt.style.use(plot_style)

          # make a big square figure
          plt.figure(figsize=(7, 7))

          # Now make each of the plots!

          # first is top left
          plt.subplot(221)
          plt.plot(my_x, y_lin, label = 'linear');
          plt.xlabel('x', fontsize = axisLabelSize);
          plt.ylabel('x', fontsize = axisLabelSize);
          plt.title('Linear', fontsize = titleSize);
          plt.text(-2, 2, 'axes panel 1')

          # then top right
          plt.subplot(222)
          plt.plot(my_x, y_quad, label = 'quadratic');
          plt.xlabel('x', fontsize = axisLabelSize);
          plt.ylabel('x squared', fontsize = axisLabelSize);
          plt.title('Quadradic', fontsize = titleSize,);
          plt.text(-1, 10, 'axes panel 2')

          # third is bottom left
          plt.subplot(223)
          plt.plot(my_x, y_cube, label = 'cubic');
          plt.xlabel('x', fontsize = axisLabelSize);
          plt.ylabel('x cubed', fontsize = axisLabelSize);
          plt.title('Cubic', fontsize = titleSize,);
          plt.text(-2, 40, 'axes panel 3')

          # then bottom right
          plt.subplot(224)
          plt.plot(my_x, y_quart, label = 'quartic');
          plt.xlabel('x', fontsize = axisLabelSize);
          plt.ylabel('x to the fourth', fontsize = axisLabelSize);
          plt.title('Quartic', fontsize = titleSize,);
          plt.text(-1, 150, 'axes panel 4')
```

```
    # finally, make everything automagically fit
    plt.tight_layout()
```

```
[55]: my_spiffy_plotting_func(my_x, y_lin, y_quad, y_cube, y_quart, 14, 12,␣
        ↪'fivethirtyeight')
```


Linear
axes panel 1
x
x
Quadradic
axes panel 2
x squared
x
Cubic
axes panel 3
x cubed
x
Quartic
axes panel 4
x to the fourth
x