An Introduction to Lambda Functions

Note: You will gain the greatest benefit from this document if you follow along and copy the code examples and execute them as you read along. Although nothing needs to be turned in at the end of this exercise, it will help you practice coding and you will better understand the need for lambda functions and their use.

A Very Simple Model

Suppose you have a model, $y = x^2$, and you have written a function to calculate the value of y in this model at any point x. You have called it `my_x2_model` and implemented it in the file `my_models.py`. The function definition is:

```python
def my_x2_model(x):
    return x ** 2
```

The derivative of the model, $\frac{dy}{dx}$ is $\frac{dy}{dx} = 2x$ and is implemented as `my_deriv_x2_model` in the same file:

```python
def my_deriv_x2_model(x):
    return 2 * x
```

A Test Script

The following script can be used to test this model:

```python
import numpy as np
import matplotlib.pyplot as plt

from my_models import *

def test_my_x2():

    # Define the range of x-values to plot
    x_min = -5
    x_max = 5
    num_values = 50
    x = np.linspace(x_min, x_max, num_values)

    # Calculate the y-values to plot
    y = my_x2_model(x)

    # Calculate the y-values of the derivative to plot
    y_prime = my_deriv_x2_model(x)

    # Plot the model values
    plt.subplot(121)
    plt.plot(x, y, 'k-')
    plt.title('Original Model')
    plt.ylabel('y-values')
    plt.xlabel('x-values')

    # Plot the derivative
    plt.subplot(122);
    plt.plot(x, y_prime, 'k-');
    plt.title(r'Exact Derivative of $y=x^2$')
    plt.ylabel('Derivative')
    plt.xlabel('x-values')

    plt.show()
```

Test Script Description

We first import modules containing functions and code that will be used. `numpy` provides numerical arrays and `matplitlib.pyplot` the plotting functionality. `my_models` contains the functions shown above.

```python
import numpy as np
import matplotlib.pyplot as plt

from my_models import *
```

Next, we create an evenly-spaced array of `num_values` values, stored in the variable `x`, between the value stored in the variable x_min and the variable x_max. This creates an array of 50 values between -5 and +5.

```
    # Define the range of x-values to plot
    x_min = -5
    x_max = 5
    num_values = 50
    x = np.linspace(x_min, x_max, num_values)
```

We then call the function my_x2_model using as the input the array x that was just created and store the result in the variable y. y is an array of 50 values containing x^2 for each of the values in the array x.

```
    # Calculate the y-values to plot
    y = my_x2_model(x)
```

my_deriv_x2_model is then called using as the input the array x, storing the result in the variable y_prime. This is an array of 50 values containing $2 * x$ for each of the values in the array x.

```
    # Calculate the y-values of the derivative to plot
    y_prime = my_deriv_x2_model(x)
```

Finally, a plot of y vs. x is created in the first subplot specified by the array (121) (see the subplot command for more information on the placement of this subplot). This should plot a parabola in the upper half of the figure. The derivative y_prime is plotted vs. x in the second subplot (122). See Figure 1 for the output of plt.show().

```
    # Plot the derivative
    plt.subplot(122);
    plt.plot(x, y_prime, 'k-');
    plt.title(r'Exact Derivative of $y=x^2$')
    plt.ylabel('Derivative')
    plt.xlabel('x-values')

    plt.show()
```

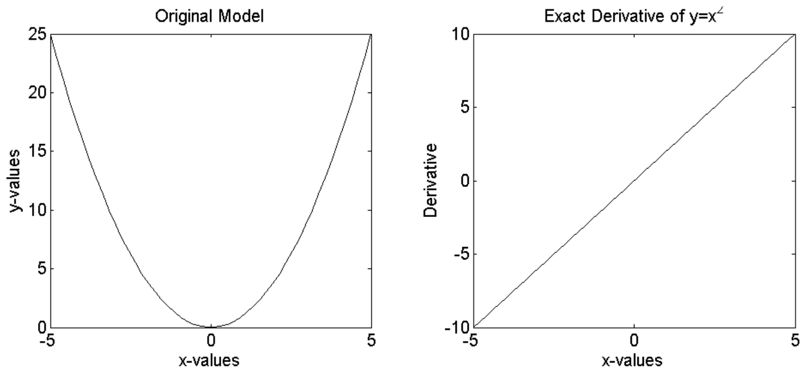

Figure 1: Results of the test script using the first model

A Few More Models

Suppose you have two more models, $y = x^3$ and $y = x^4$, that you want to represent with Python functions. We can write these just like `my_x2_model` above (contained in the same `my_models.py` file):

```python
def my_x3_model(x):
    return x ** 3
```

and

```python
def my_x4_model(x):
    return x ** 4
```

Then, we can create derivative functions for these two as well:

```python
def my_deriv_x3_model(x):
    return 3 * x ** 2
```

and

```python
def my_deriv_x4_model(x):
    return 4 * x ** 3
```

Since these models are all fairly trivial, storing them in functions this way might seem to needlessly complicates the situation, and it definitely doesn't save space. However, imagine if your model were something much more complicated, such as $y = 5e^{-\frac{1}{2}x} \sin(0.1x)$. Imagine typing that formula each time you wanted to use it. Then imagine typing the derivative of that formula each time you wanted to use it! It would be much simpler to type `my_model(x)` and `my_deriv_model(x)` each time.

Not only that, imagine if you were trying to debug your function, and you realized that, instead of $y = x^4$, your model should be $y = x^4 + 5x^3$. Now imagine a program, several hundred lines long, where you have referred to your model a dozen or more times. You now need to hunt through your code and change the model every time you used it. And if you forget to change one time, you'll leave a mistake in your code that you might not catch for days or weeks.

One more potential problem: Suppose you have finished with your project and moved on. Now, someone else comes along and wants to pick up where you left off. (This might not seem like a big deal for your final project in this class, but we had several students drop out in the middle of the semester last year, and they left their teams with poorly-documented code, which left them unable to prepare for presentations, and sometimes for the competition.) If you have saved these models in functions, you can easily see that, every time you call them, you are referring to your model, rather than just some equation x^2. Putting your models into functions makes your code more readable.

Other Models - Script Changes

The test function can be modified very simply to perform calculations using either of these models. The test function `test_my_x3` is identical to `test_my_x2`, except that `my_x3_model` and `my_deriv_x3_model` are called:

```python
def test_my_x3():

    # Define the range of x-values to plot
    x_min = -5
    x_max = 5
    num_values = 50
    x = np.linspace(x_min, x_max, num_values)

    # Calculate the y-values to plot
    y = my_x3_model(x)

    # Calculate the y-values of the derivative to plot
    y_prime = my_deriv_x3_model(x)

    # Plot the model values
    plt.subplot(121)
    plt.plot(x, y, 'k-')
    plt.title('Original Model')
    plt.ylabel('y-values')
    plt.xlabel('x-values')

    # Plot the derivative
    plt.subplot(122);
    plt.plot(x, y_prime, 'k-');
    plt.title(r'Exact Derivative of $y=x^3$')
    plt.ylabel('Derivative')
    plt.xlabel('x-values')

    plt.show()
```

Make sure to change plot labels, as was done here – not doing so can result to misleading and incorrectly labeled plots.

The results of the x3 model are shown Figure 2.

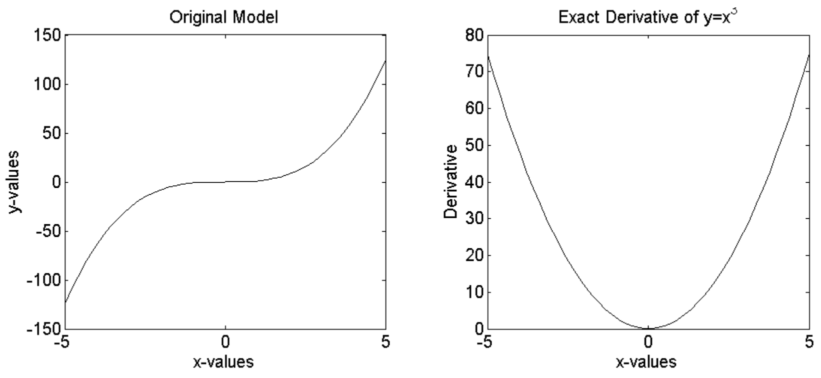

Figure 2: Changing the test script to use the second model

Approximate Derivatives - Adding Complexity

Now, sometimes, we might not care so much about the exact value of the derivative. It may be enough to get an approximation. We will discuss some approximations for derivatives later in this class, but you should already be familiar with one from your calculus classes. Recall that the derivative $\frac{dy}{dx}$ represents the slope of the curve at a given point x. You should also remember the formula for the slope m of a straight line, $m = \frac{y_1 - y_0}{x_1 - x_0}$. If we take two points (x_0, y_0) and (x_1, y_1) that lie somewhere along our function, we can approximate the derivative using the slope of the line between those two points:

$$\frac{dy}{dx} = \frac{y_1 - y_0}{x_1 - x_0} \tag{1}$$

Let's rewrite our derivative function for $y = x^2$ using this approximation for the derivative:

```python
def approximate_deriv_x2_model(x0, x1):
    # Calculate the values of y at x0 and x1
    y0 = x0 ** 2
    y1 = x1 ** 2

    # Calculate the numerator and denominator of the derivative
    numerator = y1 - y0
    denominator = x1 - x0
    if (denominator == 0):
        raise ValueError('Cannot divide by zero!')

    # Calculate the derivative
    return numerator / denominator
```

Notice that we now need two inputs for the function: x_0 and x_1. This is because we are no longer calculating the derivative at a *single* point, but approximating it as the slope of the line between *two* points. Although we don't technically need to write out the lines calculating y_0 and y_1, we have done so to make things more clear. We have also added an `if` statement to make sure that the function is not going to divide by zero. This is good programming practice. Now, let's rewrite our derivative function for the other two models using this approximation:

```python
def approximate_deriv_x3_model(x0, x1):

    # Calculate the values of y at x0 and x1
    y0 = x0 ** 3
    y1 = x1 ** 3

    # Calculate the numerator and denominator of the derivative
    numerator = y1 - y0
    denominator = x1 - x0
    if (denominator == 0):
        raise ValueError('Cannot divide by zero!')

    # Calculate the derivative
    return numerator / denominator
```

and

```
def approximate_deriv_x4_model(x0, x1):

    # Calculate the values of y at x0 and x1
    y0 = x0 ** 4
    y1 = x1 ** 4

    # Calculate the numerator and denominator of the derivative
    numerator = y1 - y0
    denominator = x1 - x0
    if (denominator == 0):
        raise ValueError('Cannot divide by zero!')

    # Calculate the derivative
    return numerator / denominator
```

Approximate Derivatives - Script Changes

Obviously, since our derivative functions now require two inputs, we need to make slight changes to the script. There are several ways we could choose the "second" set of points to send to the derivative function. One way is to add a small value (Δx), called a perturbation, to our array of x-values, and use this new array as the second input to our function, as shown here:

```
    # Calculate the y-values of the derivative to plot
    delta_x = 0.1;
    y_prime = approximate_deriv_x2_model(x,x+delta_x)
```

(The title of the plot is also changed, but no other changes need to be made.) The results should now look like Figure 3.

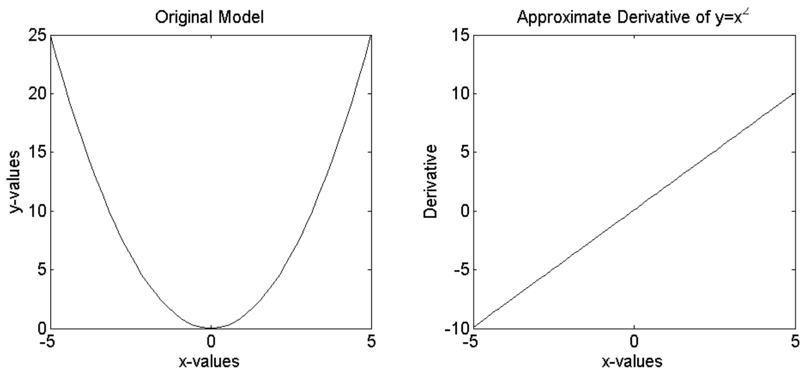

Figure 3: Results from the first model using the approximate derivative

Lambda Functions - Eliminating Repeated Code

You might be looking at what we've done so far and thinking, "There's a lot of repeated code between those three `approximate_deriv` functions. Do I really have to write a new function every time I want to take the derivative of a new model?" Of course not! Python lets us use a type of variable called a *Lambda function* to simplify this entire process. A Lambda function allows you to store the name of any function in a variable and pass it to another function. This is a very powerful tool that lets us write code that can be used in more than one situation. Instead of writing a new derivative function for every new model, we can write one derivative function and run it on any new model. Here's how we would rewrite our `approximate_deriv` function to use a Lambda functions:

```python
def approximate_deriv(fcn, x0, x1):

    # Calculate the values of y at x0 and x1
    y0 = fcn(x0);
    y1 = fcn(x1);

    # Calculate the numerator and denominator of the derivative
    numerator = y1 - y0
    denominator = x1 - x0
    if (denominator == 0):
        raise ValueError('Cannot divide by zero!')

    # Calculate the derivative
    return numerator / denominator
```

Notice that there are now three inputs instead of two. The first input, `fcn`, looks just like the name of any other variable. However, the way we use it in the function tells Python that it is a function that takes one input and has one output.

The second thing you should notice is that the Lambda functions is used in the two lines where we calculate the y-values. Instead of specifically writing out the formula for $y = f(x)$, we use the variable name `fcn` to tell Python to use the function stored in that variable to determine the values for y_1 and y_0.

Lambda Functions - Script Changes

Again, the changes we need to make to the script are only minor. We now need to create a Lambda function and pass it to the derivative function as the first input, like this:

```
16      # Set up a lambda function
17      fcn = lambda x: my_x2_model(x)
18
19      # Calculate the y-values to plot
20      y = fcn(x)
21
22      # Calculate the y-values of the derivative to plot
23      delta_x = 0.1;
24      y_prime = approximate_deriv(fcn, x, x+delta_x)
```

We've also changed the title of the plot, but, again, no other changes need to be made. The results should now look like Figure 4.

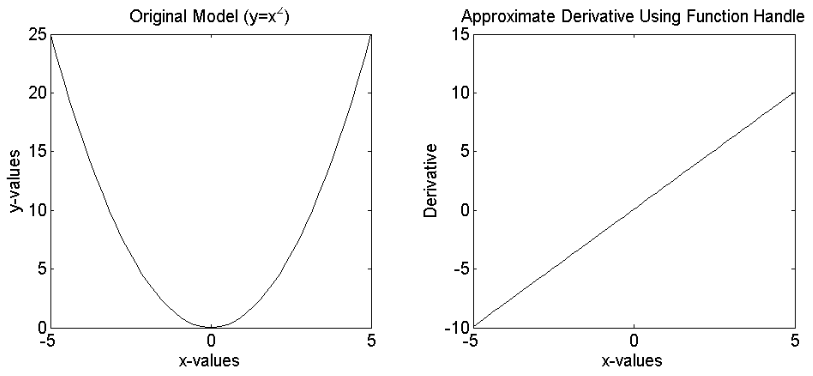

Figure 4: Results from the first model using the Lambda function form of the derivative

Notice that we use the Lambda function twice: First to calculate the y-values, and again to send to the `approximate_deriv` function to calculate the derivative values. To change the script to run a different function, all you need to do is change line 14, the declaration of the Lambda function. Lines 17 and 21 do not change! The variable `fcn` remains the same and is now used in place of our function. This produces the plot in Figure 5 for `my_x3_model`.

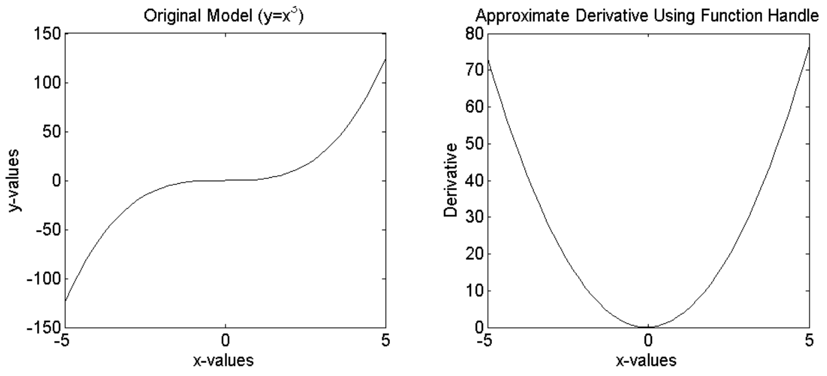

Figure 5: Results from the second model using the Lambda function form of the derivative

Summary

So what's the point of all this? Hopefully you've seen how Lambda functions can help make coding something simple like an approximate derivative simpler. By now, you've used Lambda functions in your Euler's Method function that you wrote for Lab 1. Imagine if, every time you wanted to solve an ODE to model the behavior of a system, you had to rewrite your Euler's Method code and substitute in the specific details of the model! At best, it would be a waste of time and effort. You might introduce errors into your code that you'd need to debug. If, on the other hand, you use Lambda functions to represent the models, you only have to worry about writing correct code to represent the new model every time—you know that your code for Euler's Method already works!

Throughout this semester, you'll use Lambda functions in many other assignments, which will allow the functions you write for your toolbox to be used in a wide variety of applications. Rather than working only on the specific application that you originally wrote them for, you can use these functions in other ways, on future assignments, in classes you'll take in the next few years, and hopefully throughout your career as an engineer!

Examples

Lambda functions can be used in a variety of ways in Python. In the previous examples, the Lambda function was used purely as a pass-through to another named function, accepting the same arguments as the function it is replacing. In python, functions are first-class citizens, so, a Lambda function is not strictly necessary. The same behavior can be obtained instead by simply assigning the variable `fcn` to the function:

```python
fcn = my_x2_model
```

Lambda functions become very useful when you need to create a pass-through to a named function that takes a different number of arguments. An example of this is if you have a function such as the `colebrook_equation` from Lab #3, which has four inputs and one output. For the lab assignment, we needed an interface taking one input and having one output, so we created a Lambda function. In this case, we had to define the parameters `e`, `D` and `Re` first, and then use the command

```python
fcn = lambda x: colebrook_equation(x, e, D, Re)
```

to create the Lambda function.

Another use case, involving a Lambda function with multiple inputs, is shown here. Suppose you have an ODE to simulate the motion of a train, called `train_motion`. The function has three inputs: (1) the current position, (2) the mass of the train, and (3) the current time. Now, suppose you want to use the function in an ODE simulator such as the `euler_method` function you wrote for Lab 1. Your `euler_method` function requires the ODE function to have only two inputs, the time and the position, so you can either throw up your hands in frustration and rewrite the `train_motion` fuction, or redefine it using a Lambda function, like this:

```python
train_mass = 13.5 # kg
fcn = lambda t, x: train_motion(x, train_mass, t)
```