Derivatives of Linear Functions Lab

Introduction: Start here

In this lab, we will practice our knowledge of derivatives. Remember that our key formula for derivatives, is $f'(x) = \frac{\Delta y}{\Delta x} = \frac{f(x+\Delta x)-f(x)}{\Delta x}$. So in driving towards this formula, we will do the following:

1. Learn how to represent linear and nonlinear functions in code.
2. Then because our calculation of a derivative relies on seeing the output at an initial value and the output at that value plus delta x, we need an `output_at` function.
3. Then we will be able to code the Δf function that sees the change in output between the initial x and that initial x plus the Δx
4. Finally, we will calculate the derivative at a given x value, `derivative_at` .

Learning objectives

For this first section, you should be able to answer all of the question with an understanding of our definition of a derivative:

1. Our intuitive explanation that a derivative is the instantaneous rate of change of a function
2. Our mathematical definition is that

$$f'(x) = \frac{\Delta y}{\Delta x} = \frac{f(x+\Delta x)-f(x)}{\Delta x}$$

Let's begin: Starting with functions

1. Representing Functions

We are about to learn to take the derivative of a function in code. But before doing so, we need to learn how to express any kind of function in code. This way when we finally write our functions for calculating the derivative, we can use them with both linear and nonlinear functions.

For example, we want to write the function $f(x) = 2x^2 + 4x - 10$ in a way that allows us to easily determine the exponent of each term.

This is our technique: write the formula as a list of tuples.

> A tuple is a list whose elements cannot be reassigned. But everything else, for our purposes, is the same.
>
> ```
> tuple = (7, 3)
> tuple[0] # 7
> tuple[1] # 3
> ```

> We get a TyperError if we try to reassign the tuple's elements. ```python tuple[0] = 7

TypeError: 'tuple' object does not support item assignment

```

Take the following function as an example:

$$f(x) = 4x^2 + 4x - 10$$

Here it is as a list of tuples:

In [1]:
```
four_x_squared_plus_four_x_minus_ten = [(4, 2), (4, 1), (-10, 0)]
```

So each tuple in the list represents a different term in the function. The first element of the tuple is the term's constant and the second element of the tuple is the term's exponent. Thus $4x^2$ translates to `(4, 2)` and $-10$ translates to `(-10, 0)` because $-10$ is the same as $-10 * x^0$.

> We'll refer to this list of tuples as "list of terms", or `list_of_terms` .

Ok, so give this a shot. Write $f(x) = 4x^3 + 11x^2$ as a list of terms. Assign it to the variable `four_x_cubed_plus_eleven_x_squared` .

In [2]:
```
four_x_cubed_plus_eleven_x_squared = [(4, 3), (11, 2)]
```

**2. Evaluating a function at a specific point**

Now that we can represent a function in code, let's write a Python function called `term_output` that can evaluate what a single term equals at a value of $x$.

- For example, when $x = 2$, the term $3x^2 = 3 * 2^2 = 12$.
- So we represent $3x^2$ in code as `(3, 2)` , and:
- `term_output((3, 2), 2)` should return 12

In [3]:
```
def term_output(term, input_value):
    #return term[0]*term[1]**input_value
    return term[0]*pow(input_value,term[1])
    pass
```

In [4]:
```
term_output((3, 2), 2) # 12
```

Out[4]:
```
12
```
```

Hint: To raise a number to an exponent in python, like 3^2 use the double star, as in:

```
3**2 # 9
```

Now write a function called `output_at` , when passed a `list_of_terms` and a value of x, calculates the value of the function at that value.

- For example, we'll use `output_at` to calculate $f(x) = 3x^2 - 11$.
- Then `output_at([(3, 2), (-11, 0)], 2)` should return $f(2) = 3 * 2^2 - 11 = 1$

In [5]:
```
def output_at(list_of_terms, x_value):
    results = []
    ftn_val = []
    counter = 0
    for term in list_of_terms:
        counter += 1
        if (counter < len(list_of_terms)):
            #results.append(term[0]*(x_value**term[1]))
            results.append(term[0]*pow(x_value,term[1]))
        else:
            ftn_val = list(map(lambda x:x+term[0],results))
    return ftn_val[0]
    pass
```

In [6]:
```
three_x_squared_minus_eleven = [(3, 2), (-11, 0)]
print(output_at(three_x_squared_minus_eleven, 2)) # 1
print(output_at(three_x_squared_minus_eleven, 3)) # 16
```

```
1
16
```

Now we can use our `output_at` function to display our function graphically. We simply declare a list of `x_values` and then calculate `output_at` for each of the `x_values` .

In [7]:
```
import plotly
from plotly.offline import iplot, init_notebook_mode
init_notebook_mode(connected=True)

from graph import plot, trace_values

x_values = list(range(-30, 30, 1))
y_values = list(map(lambda x: output_at(three_x_squared_minus_eleven, x),x_

three_x_squared_minus_eleven_trace  = trace_values(x_values, y_values, mode
plot([three_x_squared_minus_eleven_trace], {'title': '3x^2 - 11'})
```

3x^2 - 11
2500
2000
1500

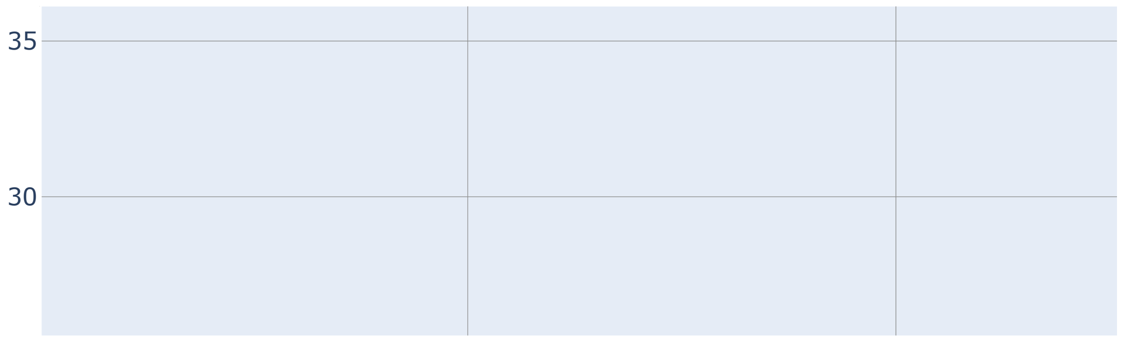
Moving to derivatives of linear functions

Let's start with a function, $f(x) = 4x + 15$. We represent the function as the following:

In [8]:
```
four_x_plus_fifteen = [(4, 1), (15, 0)]
```

We can plot the function by calculating outputs at a range of x values. Note that we use our `output_at` function to calculate the output at each individual x value.

In [9]:
```python
import plotly
from plotly.offline import iplot, init_notebook_mode
init_notebook_mode(connected=True)

from graph import plot, trace_values, build_layout

x_values = list(range(0, 6))
# layout = build_layout(y_axis = {'range': [0, 35]})


four_x_plus_fifteen_values = list(map(lambda x: output_at(four_x_plus_fifte
four_x_plus_fifteen_trace = trace_values(x_values, four_x_plus_fifteen_valu
plot([four_x_plus_fifteen_trace])
```



Ok, time for what we are here for, derivatives. Remember that the derivative is the instantaneous rate of change of a function, and is expressed as:

$$f'(x) = \frac{\Delta f}{\Delta x} = \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

Writing a function for Δf

We can see from the formula above that $\Delta f = f(x + \Delta x) - f(x)$. Write a function called `delta_f` that, given a `list_of_terms` , an `x_value` , and a value Δx, returns the change in the output over that period.

> **Hint** Don't forget about the `output_at` function. The `output_at` function takes a list of terms and an x value and returns the corresponding output. So really **`output_at` is equivalent to** $f(x)$, provided a function and a value of x.

In [10]:
```
four_x_plus_fifteen = [(4, 1), (15, 0)]
```

In [11]:
```
def delta_f(list_of_terms, x_value, delta_x):
    fx_list = []
    df_list = []
    counter = 0
    for i in range(0,len(list_of_terms)):
        fx_list.append(output_at(list_of_terms, x_value-delta_x*counter))
        counter += 1
    print(fx_list)
    fx_list.reverse()
    print(fx_list)
    for i in range(0,len(fx_list)):
        if (i < len(fx_list)-1):
            df_list.append(fx_list[i+1]-fx_list[i])
    print(len(df_list))
    print(df_list)
    return df_list[0]
    pass
```

In [12]:
```
delta_f(four_x_plus_fifteen, 2, 1) # 4
```

```
[23, 19]
[19, 23]
1
[4]
```

Out[12]:
```
4
```

So for $f(x) = 4x + 15$, when x = 2, and $\Delta x = 1$, Δf is 4.

Plotting our function, delta f, and delta x

Let's show Δf and Δx graphically.

In [13]:
```
def delta_f_trace(list_of_terms, x_value, delta_x):
    initial_f_value = output_at(list_of_terms, x_value)
    delta_f_value = delta_f(list_of_terms, x_value, delta_x)
    if initial_f_value and delta_f_value:
        trace =  trace_values(x_values=[x_value + delta_x, x_value + delta_
                              y_values=[initial_f_value, initial_f_value +
                              name = 'delta f = ' + str(delta_f_value)) # b
        return trace
```

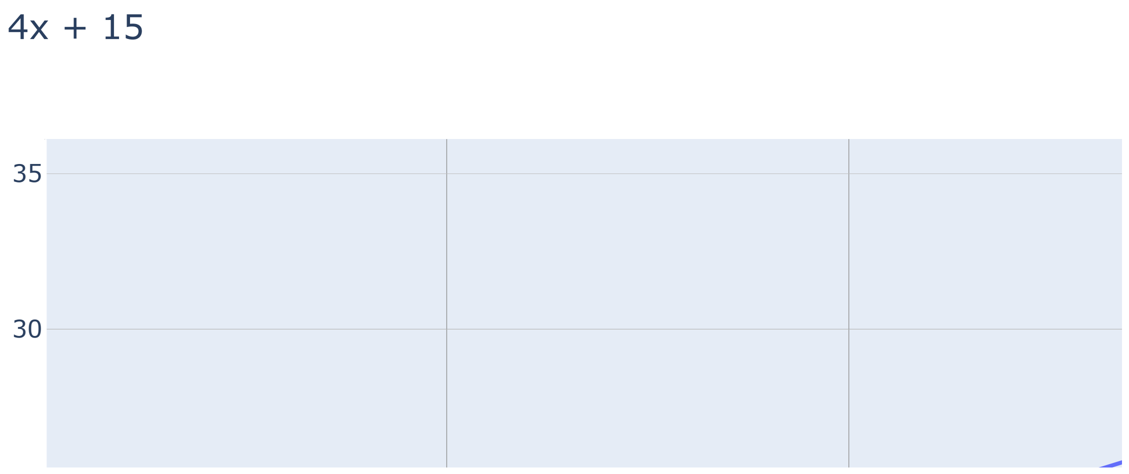
```
In [14]: trace_delta_f_four_x_plus_fifteen = delta_f_trace(four_x_plus_fifteen, 2, 1
```

```
[23, 19]
[19, 23]
1
[4]
```

Let's add another function that shows the delta x.

```
In [15]: def delta_x_trace(list_of_terms, x_value, delta_x):
             initial_f_value = output_at(list_of_terms, x_value)
             if initial_f_value:
                 trace = trace_values(x_values=[x_value, x_value + delta_x],
                                     y_values=[initial_f_value, initial_f_value], mc
                                     name = 'delta x = ' + str(delta_x))
                 return trace
```

```
In [16]: from graph import plot, trace_values

         trace_delta_x_four_x_plus_fifteen = delta_x_trace(four_x_plus_fifteen, 2, 1
         if four_x_plus_fifteen_trace and trace_delta_f_four_x_plus_fifteen and trac
             plot([four_x_plus_fifteen_trace, trace_delta_f_four_x_plus_fifteen, tra
```



Calculating the derivative

Write a function, `derivative_at` that calculates $\frac{\Delta f}{\Delta x}$ when given a `list_of_terms`, an `x_value` for the value of (x) the derivative is evaluated at, and `delta_x`, which represents Δx.

Let's try this for $f(x) = 4x + 15$. Round the result to three decimal places.

In [17]:
```
def derivative_of(list_of_terms, x_value, delta_x):
    delta_f_value = delta_f(list_of_terms, x_value, delta_x)
    return delta_f_value/delta_x
    pass
```

In [18]:
```
derivative_of(four_x_plus_fifteen, 3, 2) # 4.0
```

```
[27, 19]
[19, 27]
1
[8]
```

Out[18]:
```
4.0
```

We do: Building more plots

Ok, now that we have written a Python function that allows us to plot our list of terms, we can write a function that called `derivative_trace` that shows the rate of change, or slope, for the function between initial x and initial x plus delta x. We'll walk you through this one.

In [19]:
```
def derivative_trace(list_of_terms, x_value, line_length = 4, delta_x = .01
    derivative_at = derivative_of(list_of_terms, x_value, delta_x)
    y = output_at(list_of_terms, x_value)
    if derivative_at and y:
        x_minus = x_value - line_length/2
        x_plus = x_value + line_length/2
        y_minus = y - derivative_at * line_length/2
        y_plus = y + derivative_at * line_length/2
        return trace_values([x_minus, x_value, x_plus],[y_minus, y, y_plus]
```

> Our `derivative_trace` function takes as arguments `list_of_terms`, `x_value`, which is where our line should be tangent to our function, `line_length` as the length of our tangent line, and `delta_x` which is our Δx.

> The return value of `derivative_trace` is a dictionary that represents tangent line at that values of x. It uses the `derivative_of` function you wrote above to calculate the slope of the tangent line. Once the slope of the tangent is calculated, we stretch out this tangent line by the `line_length` provided. The beginning x value is just the midpoint minus the `line_length/2` and the ending x value is midpoint plus the `line_length/2` . Then we calculate our y endpoints by starting at the y along the function, and having them ending at `line_length/2*slope` in either direction.

In [20]:
```
tangent_line_four_x_plus_fifteen = derivative_trace(four_x_plus_fifteen, 2,
tangent_line_four_x_plus_fifteen
```

```
[23.0, 22.96]
[22.96, 23.0]
1
[0.03999999999999915]
```

Out[20]:
```
{'x': [0.0, 2, 4.0],
 'y': [15.000000000000017, 23, 30.999999999999983],
 'mode': 'lines',
 'name': "f' (x) = 3.9999999999999147",
 'text': []}
```

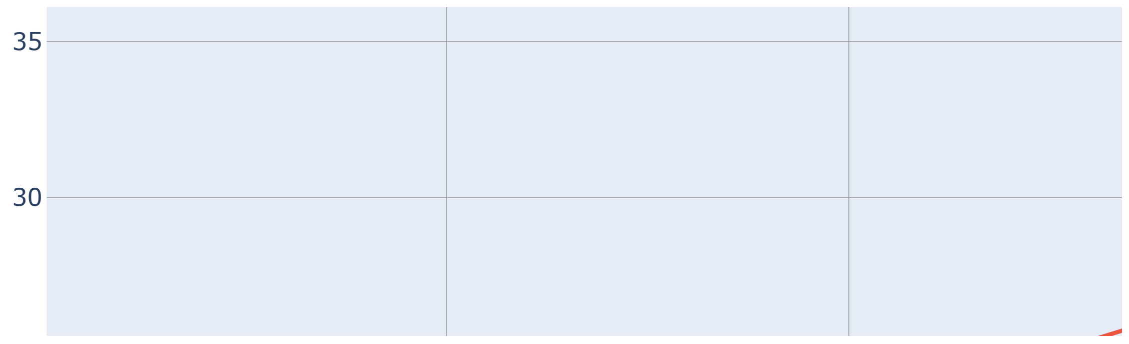
Now we provide a function that simply returns all three of these traces.

In [21]:
```
def delta_traces(list_of_terms, x_value, line_length = 4, delta_x = .01):
    tangent = derivative_trace(list_of_terms, x_value, line_length, delta_x
    delta_f_line = delta_f_trace(list_of_terms, x_value, delta_x)
    delta_x_line = delta_x_trace(list_of_terms, x_value, delta_x)
    return [tangent, delta_f_line, delta_x_line]
```

Below we can plot our trace of the function as well

In [22]:
```
delta_x = 1

# derivative_traces(list_of_terms, x_value, line_length = 4, delta_x = .01)

three_x_plus_tangents = delta_traces(four_x_plus_fifteen, 2, line_length= 2

# only plot the list of traces, if three_x_plus_tangents, does not look lik
if list(filter(None.__ne__, three_x_plus_tangents)):
    plot([four_x_plus_fifteen_trace, *three_x_plus_tangents])
```

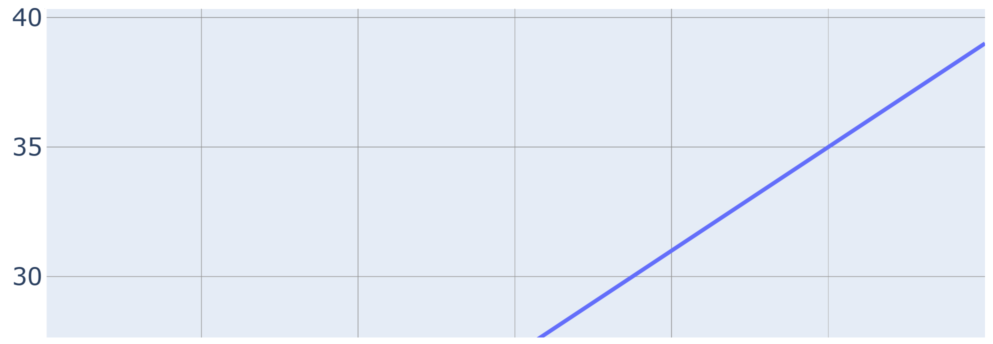
```
[23, 19]
[19, 23]
1
[4]
[23, 19]
[19, 23]
1
[4]
```



So that function highlights the rate of change is moving at precisely the point x = 2. Sometimes it is useful to see how the derivative is changing across all x values. With linear functions we know that our function is always changing by the same rate, and therefore the rate of change is constant. Let's write functions that allow us to see the function, and the derivative side by side.

In [23]:
```
from graph import make_subplots, trace_values, plot_figure

def function_values_trace(list_of_terms, x_values):
    function_values = list(map(lambda x: output_at(list_of_terms, x),x_valu
    return trace_values(x_values, function_values, mode = 'lines')

def derivative_values_trace(list_of_terms, x_values, delta_x):
    derivative_values = list(map(lambda x: derivative_of(list_of_terms, x,
    return trace_values(x_values, derivative_values, mode = 'lines')

def function_and_derivative_trace(list_of_terms, x_values, delta_x):
    traced_function = function_values_trace(list_of_terms, x_values)
    traced_derivative = derivative_values_trace(list_of_terms, x_values, de
    return make_subplots([traced_function], [traced_derivative])

four_x_plus_fifteen_function_and_derivative = function_and_derivative_trace

plot_figure(four_x_plus_fifteen_function_and_derivative)
```

```
[15, 11]
[11, 15]
1
[4]
[19, 15]
[15, 19]
1
[4]
[23, 19]
[19, 23]
1
[4]
[27, 23]
[23, 27]
1
[4]
[31, 27]
[27, 31]
1
[4]
[35, 31]
[31, 35]
1
[4]
[39, 35]
[35, 39]
1
[4]

/opt/conda/envs/learn-env/lib/python3.6/site-packages/plotly/tools.py:46
5: DeprecationWarning:

plotly.tools.make_subplots is deprecated, please use plotly.subplots.make
_subplots instead
```



Summary

In this section, we coded out our function for calculating and plotting the derivative. We started with seeing how we can represent different types of functions. Then we moved onto writing the `output_at` function which evaluates a provided function at a value of x. We calculated `delta_f` by subtracting the output at initial x value from the output at that initial x plus delta x. After calculating `delta_f` , we moved onto our `derivative_at` function, which simply divided `delta_f` from `delta_x` .

In the final section, we introduced some new functions, `delta_f_trace` and `delta_x_trace` that plot our deltas on the graph. Then we introduced the `derivative_trace` function that shows the rate of change, or slope, for the function between initial x and initial x plus delta x.