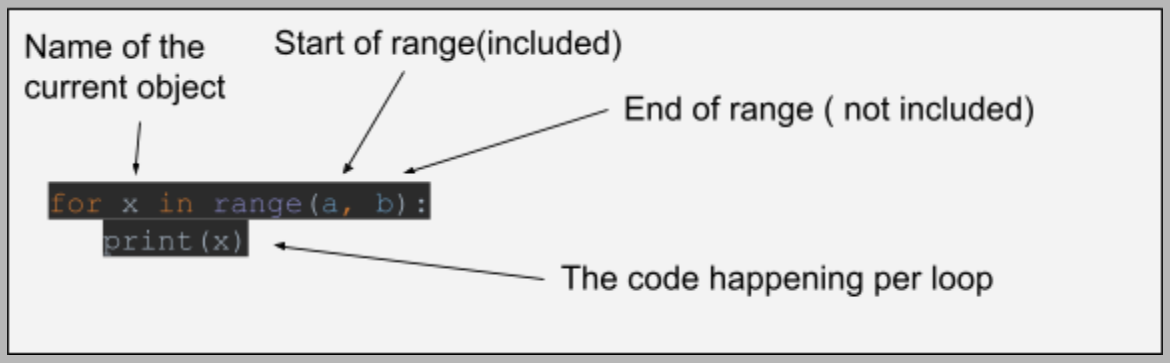
By Falk Geurten

Explanation - ex02_cfd0

Exercise 1

a.
We are going to calculate a sum using a for loop:



For that we will first define the sum and the d variable:

```
Sum = 0
d = 2*10**-4
```

Now we need to repeat the calculation d*n for 100 times with each n.

```
For n in range(0, 100+1):
        Sum += n*d
```

(fyi: x += y equals x = x+y)
(fyi: Its 100+1 and not 100, because the end isn't included in the range() argument)

---

b.
We are going to calculate the Gauss-Product of this sum:

Equation for Gauss-Product: 0.5*d*N*(N+1)

For that we will define the Gauss-Product while putting in all the variables

```
GaussProduct = 0.5*2*20**-4*100*(100+1)
```

---

c.
We are going to print out representations of the "Sum" and the "Gauss-Product":

```
print(repr(Sum))
print(repr(GaussProduct))
```

Using repr() gives you a copy of your variables converted into strings. This has the advantage of getting your results separate in a text format while not being at risk to change your original variables.

OUTPUT:

1.0099999999999998
1.01

Here we see a small difference because one is only an approximation of the actual number while the other one gives us a clear exact result.

Exercise 2

Sum equation:

$$T_N(x; x0) = \sum_{n=0}^{N} \frac{1}{n!} f^{(n)}(x0) * (x - x0)^n$$

Expanded:

$$T_N(x; x0) = f(x0) + \frac{1}{1!} f'(x0) * (x - x0)^1 + \frac{1}{2!} f''(x0) * (x - x0)^2 + ... + \frac{1}{N!} f^N(x0) * (x - x0)^N$$

Exercise 3

a.

We are going to get the Taylor series up to the 4th order for $f(x) = e^{(-2x-1)}$ at the point x0 = -0.5 in the interval x ∈ [−1, 4]:

(fyi: $f'(x) = -2e^{(-2x-1)}$ $f''(x) = 4e^{(-2x-1)}$ $f'''(x) = -8e^{(-2x-1)}$ $f^4(x) = 16e^{(-2x-1)}$

$f'(x) = -2e^{(0)}$ $f''(x) = 4e^{(0)}$ $f'''(x) = -8e^{(0)}$ $f^4(x) = 16e^{(0)}$

$f'(x) = -2$ $f''(x) = 4$ $f'''(x) = -8$ $f^4(x) = 16$)

(fyi: 0! = 1 1! = 1 2!=2 2!=6 2!=24)

First we need to make our x values in the same way we did it in ex01_cfd0:

```
x = np.linspace(-1, 4, 100)
```

(fyi:the last number (100) is not that important, it can vary and just needs to be big enough for a clear graph in the end)

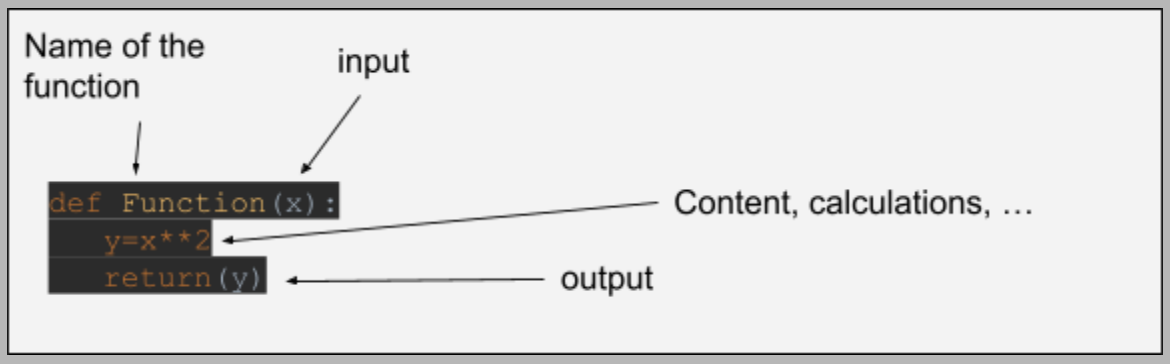
To make it easier we will first calculate all the parts separately and add them up afterwards:

```
T0 = 1
T1 = (1/1)*-2*(x- -0.5)**1
T2 = (1/2)*4*(x- -0.5)**2
T3 = (1/6)*-8*(x- -0.5)**3
T4 = (1/24)*16*(x- -0.5)**4
T = T0+T1+T2+T3+T4
```

(IMPORTANT: in this Case $T_4 = T$ $T_4 \neq T4$)

---

b.
We will now implement the answer of 3.a in a function:



Here is the function when we put in the code from 3.a:

```
def T4function(x):
    T0 = 1
    T1 = (1/1)*-2*(x- -0.5)**1
    T2 = (1/2)*4*(x- -0.5)**2
    T3 = (1/6)*-8*(x- -0.5)**3
    T4 = (1/24)*16*(x- -0.5)**4
    T = T0+T1+T2+T3+T4
    return(T)
```

---

c.
The numpy based reference function:

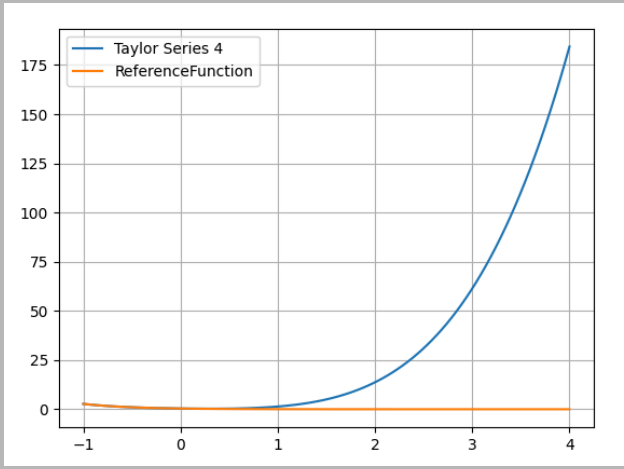
```
ReferenceFunction = np.exp(-2*x-1)
```

(fyi: in numpy e^x is np.exp(x))

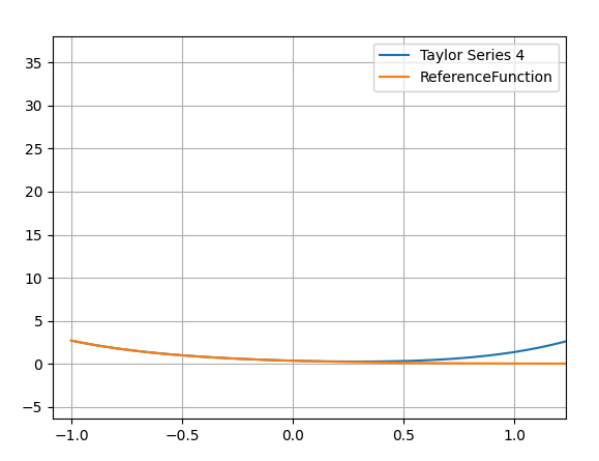
We are going to plot this graph:

```
plt.plot(x, T, label="Taylor Series 4")
plt.plot(x, ReferenceFunction, label=”Reference Function”)
plt.legend()
plt.grid()
plt.show()
```

This is how the graph should look:



Zoomed in:



smallest

Exercise 4

a.

$T_N(x; -0.5) =$

$f(-0.5) + \frac{1}{1!}f'(-0.5) * (x + 0.5) + \frac{1}{2!}f''(-0.5) * (x + 0.5)^2 + ... + \frac{1}{N!}f^N(-0.5) * (x + 0.5)^N$

$= 1 + \frac{1}{1!} - 2 * (x + 0.5) + \frac{1}{2!}4 * (x + 0.5)^2 + ... + \frac{1}{N!}(-2)^N * (x + 0.5)^N$

If we put this in a sum it will look like this:

$\sum_{n=0}^{N} \frac{1}{n!}(-2)^n * (x - x0)^n$

---

b.

This is the code for our function:

```
def factorial(n):
    if n == 1
        return n
    else:
        return n*factorial(n-1)
```

Note that this is a recursive function. That means the part of the function is the function itself.

What the code does is, it checks if the current number n is 1 and if not, it will times the current number with the process just described but with n-1 instead of 1. If at some point the “new n”=1 it will multiply everything backwards to get the final solution.

For the example of 4! or factorial(4) the code would do this:

Check if 4 = 1? No, so start again with 4-1
  Check if 3 = 1? No, so start again with 3-1
    Check if 2 = 1? No, so start again with 2-1
      Check if 2 = 1? Yes so it takes the 1:
    Times the 2 by the 1 = 2
  Times the 3 by the 2 = 6
Times the 4 by the 6 = 24

IMPORTANT: for the special case of 0! We will add this:

```
def factorial(n):
    if n == 0
        return 1
    else:
        if n == 1
            return n
        else:
            return n*factorial(n-1)
```

---

c.
For this exercise we will combine all new Elements:
First we declare our function and our variables:

```
def TaylorSeriousFunction(x_coordinates, N):
        final_function = 0
        n = 0
```

Then we will go through the following formula N times while updating the variables:

$$\sum_{n=0}^{N} \frac{1}{n!}(-2)^n * (x - x0)^n$$

```
for x in range(0, N+1):
        final_function += (1/factorial(n))*((-2)*n)*((x_coordinates - -0.5)**n)
        n+= 1
```

Putting everything together and returning the final_function:

By Falk Geurten

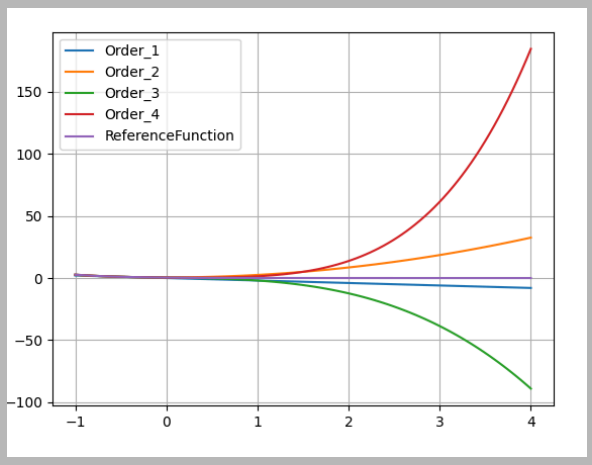
```
def TaylorSeriousFunction(x_coordinates, N):
        final_function = 0
        n = 0
        for x in range(0, N+1):
                final_function += (1/factorial(n))*((-2)*n)*((x_coordinates - -0.5)**n)
                n+= 1
        return final_function
```

---

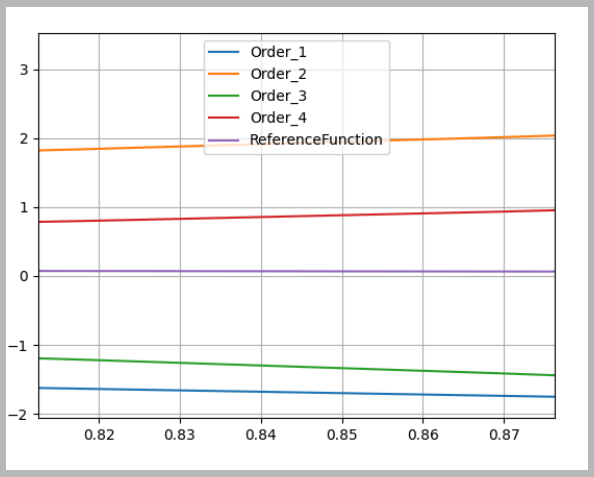
d.
We will plot the function for N = 1; 2; 3; 4

```
plt.plot(x, TaylorSeriesFunction(x, 1), label="Order_1")
plt.plot(x, TaylorSeriesFunction(x, 2), label="Order_2")
plt.plot(x, TaylorSeriesFunction(x, 3), label="Order_3")
plt.plot(x, TaylorSeriesFunction(x, 4), label="Order_4")
plt.plot(x, ReferenceFunction, label="ReferenceFunction")
plt.legend()
plt.grid()
plt.show()
```

This is how the graph should look:



Zoomed in:



We can see that the higher N, the further away are the values at distant points from x0
We can see that the higher N, the more accurate are the values close to x0.