BME646 and ECE60146: Homework 1
Spring 2025

1 Introduction

This homework is designed to strengthen your understanding of Python Object-Oriented Programming (OOP), with a particular focus on its applications in PyTorch. It is the only assignment that focuses on general OOP principles.

Future assignments will involve the Python classes from the PyTorch platform and your homework will involve either using them or extending them in the object-oriented sense.

In this homework, you'll apply OOP concepts to model exponential growth related to a hypothetical exercise in population dynamics.

2 Programming Tasks

Explanation of Task 1:

A BioModel class is created and initialized using a __init__() method with a sequence parameter.

In [1]:
```
class BioModel(object):                    # Defining base class BioModel
    def __init__ (self, sequence):         # Initializing instance with a parameter "sequence"
        self.sequence = sequence           # Instance variable sequence
```

Explanation of Task 2:

A derived class named "ExponentialGrowthModel" is created to inherit from the base class BioModel. The derived class itself takes in a start and a rate parameters. They are assigned to "self" meaning they are only valid for the instance under construction.

In [2]:
```
class ExponentialGrowthModel(BioModel):      # Defining a subclass ExponentialGrowthModel
    def __init__(self, start, rate):         # Initializing parameters "start" and "step"
        super().__init__([])                 # Initializing base class with an empty list
        self.start = start                   # Instance variable start
        self.rate = rate                     # Instance variable rate
```

Explanation of Task 3:

The derived class ExponentialGrowthModel is further expanded to be callable using the __call__() method. Whenever the class is called using a length parameter, it uses the start and rate attributes of the instance in addition to the following fomula to print out a sequence

$$next value = current value × (1 + rate)$$

A system supplied __len__() method is also implimented to return the length of the new sequence.

In [3]:
```
class ExponentialGrowthModel(BioModel):
    def __init__(self, start, rate):
        super().__init__([])
        self.start = start
        self.rate = rate

    def __call__(self, length):                                  # __call__() method takes in a length para
        self.currentvalue = self.start
        self.sequence = []                                       # An empty list is initialized to store va
        for _ in range(length):
            self.sequence.append(self.currentvalue)              # Looping length number of times and stori
            self.nextvalue = self.currentvalue * (1 + self.rate) # Next value is calculated using the growt
            self.currentvalue = self.nextvalue                   # Current value is updated to be the next
        print([round(val, 2) for val in self.sequence])          # Rounding and printing the computed seque
        return self.sequence

    def __len__(self):
        return len(self.sequence)                                # __len__() returns the length of the sequ
```

In [4]:
```
# Required outputs
GM = ExponentialGrowthModel(start=100, rate=0.1)
GM(length=5)                                    # [100, 110.0, 121.0, 133.1, 146.41]
print(len(GM))                                  # 5
```

```
[100, 110.0, 121.0, 133.1, 146.41]
5
```

In [5]:
```
# My chosen outputs
GM = ExponentialGrowthModel(start=200, rate=0.5)
GM(length=10)                                   # [200, 300.0, 450.0, 675.0, 1012.5, 1518.75, 2278.12, 3417.1
print(len(GM))                                  # 10
```

```
[200, 300.0, 450.0, 675.0, 1012.5, 1518.75, 2278.12, 3417.19, 5125.78, 7688.67]
10
```

Explanation of Task 4:

The base class BioModel is further expanded to return the length of the sequence as well as make it iterable. The __iter__() intialize an index which is then incremented by 1 the __next__() method is used. While this index smaller than the length of the sequence, the correnponding element of the sequence is returned, Otherwise a 'StopIteration' exception is raised.

In [6]:
```
class BioModel(object):
    def __init__ (self, sequence):
        self.sequence = sequence

    def __len__(self):
        return len(self.sequence)                       # __len__() returns the length of the sequence

    def __iter__(self):                                 # __iter__() method calls the iterator class BioModel
        self.idx = -1                                   # Initialize the index
        return self

    def __next__(self):                                 # __next__() is used to setup the iteration logic
        self.idx += 1                                   # Incrementing the index by 1 each time __next__() met
        if self.idx < len(self.sequence):
            return round(self.sequence[self.idx], 2)    # Returns sequence value one at a time
        else:
            raise StopIteration

class ExponentialGrowthModel(BioModel):
    def __init__(self, start, rate):
        super().__init__([])
        self.start = start
        self.rate = rate

    def __call__(self, length):
        self.currentvalue = self.start
        self.sequence = []
        for _ in range(length):
            self.sequence.append(self.currentvalue)
            self.nextvalue = self.currentvalue * (1 + self.rate)
            self.currentvalue = self.nextvalue
        print([round(val, 2) for val in self.sequence])
        return self.sequence
```

In [7]:
```
# Required outputs
GM = ExponentialGrowthModel(start=100, rate=0.1)
GM(length=5)                                        # [100, 110.0, 121.0, 133.1, 146.41]
print(len(GM))                                      # 5
print([n for n in GM])                              # [100, 110.0, 121.0, 133.1, 146.41]
```

```
[100, 110.0, 121.0, 133.1, 146.41]
5
[100, 110.0, 121.0, 133.1, 146.41]
```

In [8]:
```
# My chosen outputs
GM = ExponentialGrowthModel(start=200, rate=0.5)
GM(length=10)                                       # [200, 300.0, 450.0, 675.0, 1012.5, 1518.75, 2278.12, 34
print(len(GM))                                      # 10
print([n for n in GM])                              # [200, 300.0, 450.0, 675.0, 1012.5, 1518.75, 2278.12, 34
```

```
[200, 300.0, 450.0, 675.0, 1012.5, 1518.75, 2278.12, 3417.19, 5125.78, 7688.67]
10
[200, 300.0, 450.0, 675.0, 1012.5, 1518.75, 2278.12, 3417.19, 5125.78, 7688.67]
```

Explanation of Task 5:

A new derived class ExponentialDecayModel is defined in the same way as the ExponentialGrowthModel class with BioModel class as the base class. The only difference is that the growth equation is replaced withe the following decay equation:

$$next value = current value × (1 - rate)$$

In [9]:
```
class BioModel(object):
    def __init__ (self, sequence):
        self.sequence = sequence

    def __len__(self):
        return len(self.sequence)

    def __iter__(self):
        self.idx = -1
        return self

    def __next__(self):
        self.idx += 1
        if self.idx < len(self.sequence):
            return round(self.sequence[self.idx], 2)
        else:
            raise StopIteration

class ExponentialDecayModel(BioModel):                              # Defining a subclass ExponentialDecayModel
    def __init__(self, start, rate):
        super().__init__([])
        self.start = start
        self.rate = rate

    def __call__(self, length):
        self.currentvalue = self.start
        self.sequence = []
        for _ in range(length):
            self.sequence.append(self.currentvalue)
            self.nextvalue = self.currentvalue * (1 - self.rate)    # New decay equation is implemented
            self.currentvalue = self.nextvalue
        print([round(val, 2) for val in self.sequence])
        return self.sequence
```

In [10]:
```
# Required outputs
DM = ExponentialDecayModel(start=100, rate=0.2)
DM(length=5)                                        # [100, 80.0, 64.0, 51.2, 40.96]
print(len(DM))                                      # 5
print([n for n in DM])                              # [100, 80.0, 64.0, 51.2, 40.96]
```

```
[100, 80.0, 64.0, 51.2, 40.96]
5
[100, 80.0, 64.0, 51.2, 40.96]
```

In [11]:
```
# My chosen outputs
DM = ExponentialDecayModel(start=200, rate=0.5)
DM(length=5)                                        # [200, 100.0, 50.0, 25.0, 12.5]
print(len(DM))                                      # 5
print([n for n in DM])                              # [200, 100.0, 50.0, 25.0, 12.5]
```

```
[200, 100.0, 50.0, 25.0, 12.5]
5
[200, 100.0, 50.0, 25.0, 12.5]
```

Explanation of Task 6:

The base class BioModel is futher expaneded with the __eq__() method that is programmed to raise an exception if the lengths to the current instance's sequence and another instance's sequence do not match. Otherwise, the two sequences are paired up using the zip method and designed to return 1 if the two paired elements matches. These 1s are summed up to count the number of matching elements.

In [12]:
```
class BioModel(object):
    def __init__ (self, sequence):
        self.sequence = sequence

    def __len__(self):
        return len(self.sequence)

    def __iter__(self):
        self.idx = -1
        return self

    def __next__(self):
        self.idx += 1
        if self.idx < len(self.sequence):
            return round(self.sequence[self.idx], 2)
        else:
            raise StopIteration
```

```
    def __eq__(self, model2):                                   # __eq__() method takes in the instan
        if len(self.sequence) != len(model2.sequence):
            raise ValueError("Two arrays are not equal in length!")       # Raises an exception if the 2 sequen
        c_matchingElemets = sum(1 for seq1, seq2 in zip(self.sequence, model2.sequence) if seq1==seq2)
        return c_matchingElemets                                # Counts and returns the number of ma

class ExponentialGrowthModel(BioModel):
    def __init__(self, start, rate):
        super().__init__([])
        self.start = start
        self.rate = rate

    def __call__(self, length):
        self.currentvalue = self.start
        self.sequence = []
        for _ in range(length):
            self.sequence.append(self.currentvalue)
            self.nextvalue = self.currentvalue * (1 + self.rate)
            self.currentvalue = self.nextvalue
        print([round(val, 2) for val in self.sequence])
        return self.sequence

class ExponentialDecayModel(BioModel):
    def __init__(self, start, rate):
        super().__init__([])
        self.start = start
        self.rate = rate

    def __call__(self, length):
        self.currentvalue = self.start
        self.sequence = []
        for _ in range(length):
            self.sequence.append(self.currentvalue)
            self.nextvalue = self.currentvalue * (1 - self.rate)
            self.currentvalue = self.nextvalue
        print([round(val, 2) for val in self.sequence])
        return self.sequence
```

In [13]:

```
# Required outputs
GM = ExponentialGrowthModel(start=100, rate=0.1)
GM(length=5)                                        # [100, 110.0, 121.0, 133.1, 146.41]

GM2 = ExponentialGrowthModel(start=100, rate=0.2)
GM2(length=5)                                       # [100, 120.0, 144.0, 172.8, 207.36]

print(GM == GM2)                                    # 1

GM3 = ExponentialGrowthModel(start=100, rate=0.2)
GM3(length=3)                                       # [100, 120.0, 144.0]

print(GM == GM3)                                    # ValueError: Two arrays are not equal in length!
```

```
[100, 110.0, 121.0, 133.1, 146.41]
[100, 120.0, 144.0, 172.8, 207.36]
1
[100, 120.0, 144.0]
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[13], line 13
     10 GM3 = ExponentialGrowthModel(start=100, rate=0.2)
     11 GM3(length=3)
---> 13 print(GM == GM3)

Cell In[12], line 21, in BioModel.__eq__(self, model2)
     19 def __eq__(self, model2):
     20     if len(self.sequence) != len(model2.sequence):
---> 21         raise ValueError("Two arrays are not equal in length!")
     22     c_matchingElemets = sum(1 for seq1, seq2 in zip(self.sequence, model2.sequence) if seq1==seq2)
     23     return c_matchingElemets

ValueError: Two arrays are not equal in length!
```

In [14]:

```
# My chosen outputs
GM = ExponentialGrowthModel(start=200, rate=0.5)
GM(length=5)                                         # [200, 300.0, 450.0, 675.0, 1012.5]

DM = ExponentialDecayModel(start=1012.5, rate=0.5)
DM(length=5)                                         # [1012.5, 506.25, 253.12, 126.56, 63.28]

print(GM == DM)                                      # 0

GM3 = ExponentialGrowthModel(start=200, rate=0.5)
GM3(length=10)                                       # [200, 300.0, 450.0, 675.0, 1012.5, 1518.75, 2278.12, 341

print(GM == GM3)                                     # ValueError: Two arrays are not equal in length!
```

```
[200, 300.0, 450.0, 675.0, 1012.5]
[1012.5, 506.25, 253.12, 126.56, 63.28]
0
[200, 300.0, 450.0, 675.0, 1012.5, 1518.75, 2278.12, 3417.19, 5125.78, 7688.67]
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[14], line 13
     10 GM3 = ExponentialGrowthModel(start=200, rate=0.5)
     11 GM3(length=10)
---> 13 print(GM == GM3)

Cell In[12], line 21, in BioModel.__eq__(self, model2)
     19 def __eq__(self, model2):
     20     if len(self.sequence) != len(model2.sequence):
---> 21         raise ValueError("Two arrays are not equal in length!")
     22     c_matchingElemets = sum(1 for seq1, seq2 in zip(self.sequence, model2.sequence) if seq1==seq2)
     23     return c_matchingElemets

ValueError: Two arrays are not equal in length!
```

Explanation of Task 7:

Reproducing the results using my own chosen values

In [15]:
```
print("################# Task 1, 2, 3 & 4 #################")
GM = ExponentialGrowthModel(start=300, rate=0.2)
GM(length=10)                                          # [300, 360.0, 432.0, 518.4, 622.08, 746.5, 895.8, 1074.95,
print(len(GM))                                         # 10
print([n for n in GM])                                 # [300, 360.0, 432.0, 518.4, 622.08, 746.5, 895.8, 1074.95,
print("\n")

print("###################### Task 5 ######################")
DM = ExponentialDecayModel(start=1000, rate=0.5)
DM(length=6)                                           # [1000, 500.0, 250.0, 125.0, 62.5, 31.25]
print(len(DM))                                         # 6
print([n for n in DM])                                 # [1000, 500.0, 250.0, 125.0, 62.5, 31.25]
print("\n")

print("###################### Task 6 ######################")
GM = ExponentialGrowthModel(start=200, rate=0.5)
GM(length=5)                                           # [200, 300.0, 450.0, 675.0, 1012.5]
DM = ExponentialDecayModel(start=1000, rate=0.5)
DM(length=5)                                           # [1000, 500.0, 250.0, 125.0, 62.5]
print(GM == DM)                                        # 0
GM3 = ExponentialGrowthModel(start=200, rate=0.5)
GM3(length=10)                                         # [200, 300.0, 450.0, 675.0, 1012.5, 1518.75, 2278.12, 3417.
print(GM == GM3)                                       # ValueError: Two arrays are not equal in length!
```

```
################# Task 1, 2, 3 & 4 #################
[300, 360.0, 432.0, 518.4, 622.08, 746.5, 895.8, 1074.95, 1289.95, 1547.93]
10
[300, 360.0, 432.0, 518.4, 622.08, 746.5, 895.8, 1074.95, 1289.95, 1547.93]


###################### Task 5 ######################
[1000, 500.0, 250.0, 125.0, 62.5, 31.25]
6
[1000, 500.0, 250.0, 125.0, 62.5, 31.25]


###################### Task 6 ######################
[200, 300.0, 450.0, 675.0, 1012.5]
[1000, 500.0, 250.0, 125.0, 62.5]
0
[200, 300.0, 450.0, 675.0, 1012.5, 1518.75, 2278.12, 3417.19, 5125.78, 7688.67]
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[15], line 23
     21 GM3 = ExponentialGrowthModel(start=200, rate=0.5)
     22 GM3(length=10)
---> 23 print(GM == GM3)

Cell In[12], line 21, in BioModel.__eq__(self, model2)
     19 def __eq__(self, model2):
     20     if len(self.sequence) != len(model2.sequence):
---> 21         raise ValueError("Two arrays are not equal in length!")
     22     c_matchingElemets = sum(1 for seq1, seq2 in zip(self.sequence, model2.sequence) if seq1==seq2)
     23     return c_matchingElemets

ValueError: Two arrays are not equal in length!
```

3 Bonus

Explanation of Task 1:

A new subsclass CombinedBioModel is initialized with the ExponentialGrowthModel and ExponentialDecayModel. This class takes in the growth and decay starts and rates parameters and passes them to the appropriate classes to for the respective sequences. The subclass is made callable to return and print the elementwise multiplication og the two sequences.

In [16]:
```
class BioModel(object):
    def __init__ (self, sequence):
        self.sequence = sequence

    def __len__(self):
        return len(self.sequence)

    def __iter__(self):
        self.idx = -1
        return self

    def __next__(self):
        self.idx += 1
        if self.idx < len(self.sequence):
            return round(self.sequence[self.idx], 2)
        else:
            raise StopIteration

    def __eq__(self, model2):
        if len(self.sequence) != len(model2.sequence):
            raise ValueError("Two arrays are not equal in length!")
        c_matchingElemets = sum(1 for seq1, seq2 in zip(self.sequence, model2.sequence) if seq1==seq2)
        return c_matchingElemets

class ExponentialGrowthModel(BioModel):
    def __init__(self, start, rate):
        super().__init__([])
        self.start = start
        self.rate = rate

    def __call__(self, length):
        self.currentvalue = self.start
        self.sequence = []
        for _ in range(length):
            self.sequence.append(self.currentvalue)
            self.nextvalue = self.currentvalue * (1 + self.rate)
            self.currentvalue = self.nextvalue
        return self.sequence

class ExponentialDecayModel(BioModel):
    def __init__(self, start, rate):
        super().__init__([])
        self.start = start
        self.rate = rate

    def __call__(self, length):
        self.currentvalue = self.start
        self.sequence = []
        for _ in range(length):
            self.sequence.append(self.currentvalue)
            self.nextvalue = self.currentvalue * (1 - self.rate)
            self.currentvalue = self.nextvalue
        return self.sequence

class CombinedBioModel(ExponentialGrowthModel, ExponentialDecayModel):             # CombinedBioModel subclass
    def __init__(self, growth_start, growth_rate, decay_start, decay_rate):        # It takes in growth and dec
        self.growth_start = growth_start                                           # Instance variable
        self.growth_rate = growth_rate                                             # Instance variable
        self.decay_start = decay_start                                             # Instance variable
        self.decay_rate = decay_rate                                               # Instance variable

    def __call__(self, length):
        GM = ExponentialGrowthModel(start=self.growth_start, rate=self.growth_rate)
        seqGM = GM(length)                                                         # Generates a sequence from

        DM = ExponentialDecayModel(start=self.decay_start, rate=self.decay_rate)
        seqDM = DM(length)                                                         # Generates a sequence from

        self.multiplied = [seq1*seq2 for seq1, seq2 in zip(seqGM, seqDM)]          # Elementwise multiplies the
        print([round(val, 2) for val in self.multiplied])
        return self.multiplied
```

In [17]:
```
# Required outputs
CBM = CombinedBioModel(growth_start=100, growth_rate=0.1, decay_start=1.0, decay_rate=0.05)
CBM(length=5)                                                                      # [100.0, 104.5, 109.2, 114.
```

```
[100.0, 104.5, 109.2, 114.12, 119.25]
```

Out[17]:
```
[100.0,
 104.50000000000001,
 109.20250000000003,
 114.11661250000003,
 119.25186006250004]
```

In [18]:
```
# My chosen outputs
CBM = CombinedBioModel(growth_start=200, growth_rate=0.5, decay_start=2.0, decay_rate=0.1)
CBM(length=7)                                                                    # [400.0, 540.0, 729.0, 984.
```

Out[18]:
```
[400.0, 540.0, 729.0, 984.15, 1328.6, 1793.61, 2421.38]
[400.0,
 540.0,
 729.0,
 984.1500000000001,
 1328.6025000000002,
 1793.6133750000006,
 2421.3780562500006]
```

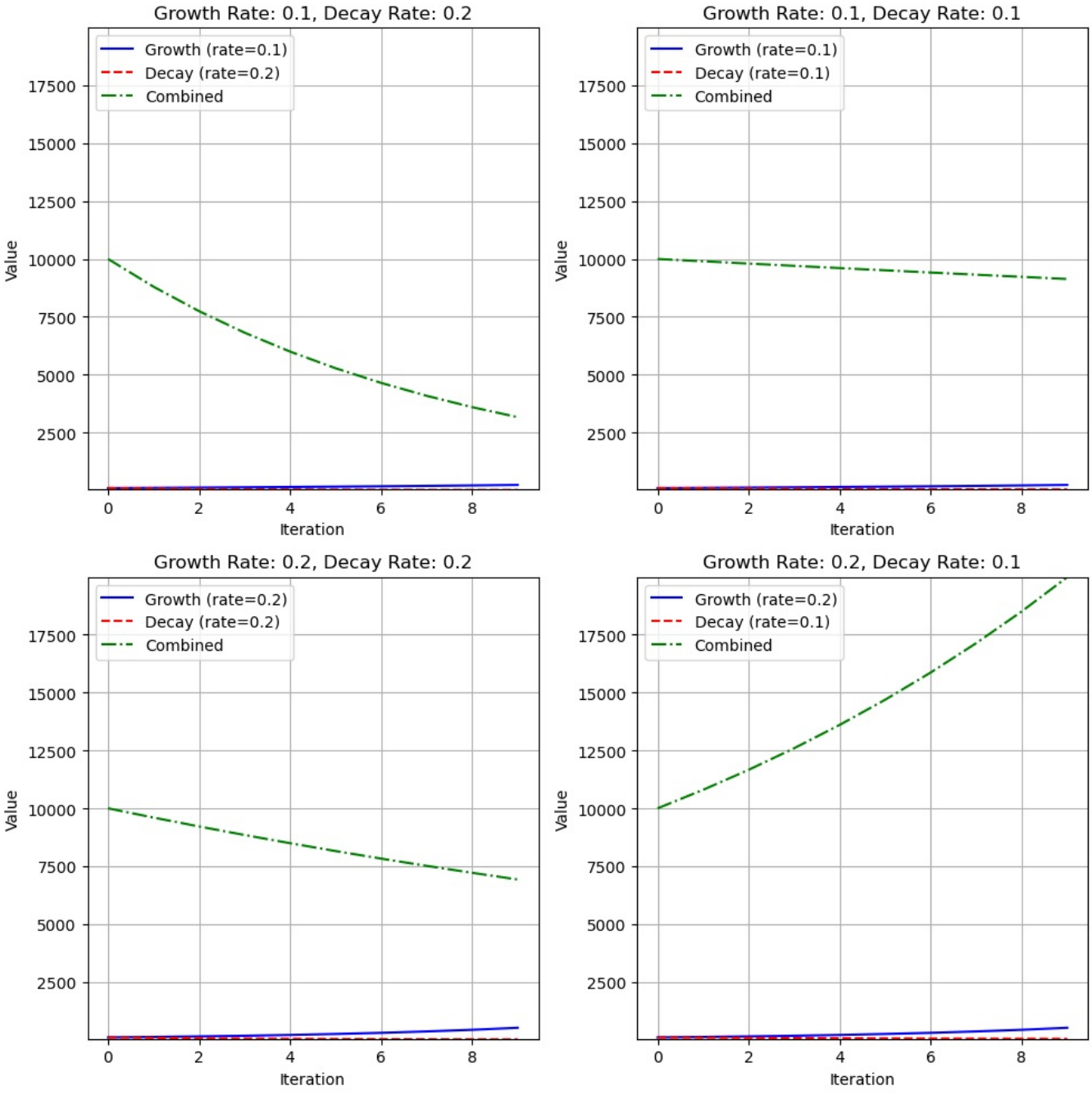
Explanation of Task: 2

By looping through a set of two growth and decay rates, sequences from ExponentialGrowthModel, ExponentialDecayModel and CombinedBioModel are derived and temorarily stored in an array. The y axis minima and maxima is determined from the decay sequence and combined sequence repectively as they demostrates the extreme values. The plotting is then setup with each distinguised with a different color and style using the matplotlib library,

In [19]:
```
import matplotlib.pyplot as plt                                                  # Importing matplotlib library
```

In [20]:
```
growth_rates = [0.1, 0.2]
decay_rates = [0.2, 0.1]
starts = [100, 100]
length = 10

arr = []

for growth_rate in growth_rates:                                                 # Looping through eacg growth and d
    for decay_rate in decay_rates:
        GM = ExponentialGrowthModel(start=starts[0], rate=growth_rate)
        seqGM = GM(length=length)

        DM = ExponentialDecayModel(start=starts[1], rate=decay_rate)
        seqDM = DM(length=length)

        CBM = CombinedBioModel(growth_start=starts[0], growth_rate=growth_rate, decay_start=starts[1], decay_ra
        seqCBM = CBM(length=length)

        arr.append([growth_rate, decay_rate, seqGM, seqDM, seqCBM])              # Storing the data in a list


fig, axes = plt.subplots(2, 2, figsize=(10, 10))                                 # Defining the plot parameters
axes = axes.flatten()
for _, _, seqGM, seqDM, seqCBM in arr: ymin, ymax = min(seqDM), max(seqCBM) # Determiming the y axis limits

for i, (growth_rate, decay_rate, seqGM, seqDM, seqCBM) in enumerate(arr):
    ax = axes[i]
    # Plotting each line with a differen color and style
    ax.plot(range(length), seqGM, label=f"Growth (rate={growth_rate})", color="blue", linestyle="solid")
    ax.plot(range(length), seqDM, label=f"Decay (rate={decay_rate})", color="red", linestyle="dashed")
    ax.plot(range(length), seqCBM, label="Combined", color="green", linestyle="dashdot")
    # Labelling each plot
    ax.set_title(f"Growth Rate: {growth_rate}, Decay Rate: {decay_rate}")
    ax.set_xlabel("Iteration")
    ax.set_ylabel("Value")
    ax.set_ylim(ymin, ymax)
    ax.legend(loc="upper left")
    # Turning plot grid
    ax.grid(True)

# Displaying the plots
plt.tight_layout()
plt.show()
```

```
[10000, 8800.0, 7744.0, 6814.72, 5996.95, 5277.32, 4644.04, 4086.76, 3596.35, 3164.78]
[10000, 9900.0, 9801.0, 9702.99, 9605.96, 9509.9, 9414.8, 9320.65, 9227.45, 9135.17]
[10000, 9600.0, 9216.0, 8847.36, 8493.47, 8153.73, 7827.58, 7514.47, 7213.9, 6925.34]
[10000, 10800.0, 11664.0, 12597.12, 13604.89, 14693.28, 15868.74, 17138.24, 18509.3, 19990.05]
```


Growth Rate: 0.1, Decay Rate: 0.2
Growth (rate=0.1)
Decay (rate=0.2)
Combined
Value
Iteration
Growth Rate: 0.1, Decay Rate: 0.1
Growth (rate=0.1)
Decay (rate=0.1)
Combined
Value
Iteration
Growth Rate: 0.2, Decay Rate: 0.2
Growth (rate=0.2)
Decay (rate=0.2)
Combined
Value
Iteration
Growth Rate: 0.2, Decay Rate: 0.1
Growth (rate=0.2)
Decay (rate=0.1)
Combined
Value
Iteration

In [ ]: