ECE60146 Homework 1

Leonard Fan

January 2025

1 Tasks

1.1 1

The task asks us to create a Biomodel class with a sequence instance variable. The code given to us in the assignment is as follows:

```
# Create class BioModel
class BioModel(object):
    def __init__(self, sequence):
        self.sequence = sequence # Class has input sequence
```

1.2 2

The task asks us to extend the BioModel class into an ExponentialGrowthModel subclass. The `__init__` method is defined with two parameters: `start` and `rate`, which serve as the start and the rate of the growth model.

```
# Create class ExponentialGrowthModel
class ExponentialGrowthModel(BioModel):
    def __init__(self, start, rate):
        super().__init__(sequence=[])
        self.start = start # Class has input start
        self.rate = rate   # Class has input rate
```

To make ExponentialGrowthModel a subclass of BioModel, we instead use BioModel in the class definition. In the `__init__` method, we define the two input parameters `start` and `rate`. The values are stored in `self.start` and `self.rate`.

1.3 3

The task is to expand `ExponentialGrowthModel` further to make its instances callable. It will include a method with a parameter `length`. A sequence of values is calculated by:

$$\text{next value} = \text{current value} \times (1 + \text{rate})$$

The instance should also print the computed sequence.

```
# Create class ExponentialGrowthModel
class ExponentialGrowthModel(BioModel):
    def __init__(self, start, rate):
        super().__init__(sequence=[])
        self.start = start # Class has input start
        self.rate = rate   # Class has input rate

    # Task 3
    # Callable method
    def __call__(self, length):
        self.sequence = [self.start] # Sequence begins with
            start parameter
        curr_val = self.start        # Calculation begins
            with start parameter

        # Loop through and calculate sequence
        for _ in range(1, length):
            next_val = curr_val * (1 + self.rate) #
                Calculate next value
            self.sequence.append(next_val)       # Add next
                value to sequence
            curr_val = next_val                  # Update
                current value to calculated value

        print(self.sequence) # Print final sequence

    # Define __len__ method to print length when called
    def __len__(self):
        return len(self.sequence)
```

The `__call__` method will run a program when the instance is called. We start with the sequence with the given `start` parameter to calculate the sequence. The variable `curr_val` is iterated through and updated using the equation defined previously. Each new calculated value is appended to the parameter `self.sequence`. Lastly, the final sequence is printed.

The example shows the `len()` function called on the instance. We must add the

`__len__` method. This method returns the length of the sequence when called.

The output is tested with the example:

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

The output is as follows:

```
[100, 110.00, 121.00, 133.10, 146.41]
5
```

Which matches the expected output in the example.

1.4 4

The task is to modify `BioModel` to be used as an integrator.

```
# Define iter method to start iterable
def __iter__(self):
    self.index = 0 # Start index at 0
    return self

# Define next method to continue iterable
def __next__(self):
    # Logic to determine when to stop iteration
    if self.index < len(self.sequence):
        self.index += 1
        return self.sequence[self.index - 1] # Subtract one
            from index for accurate output
    else:
        raise StopIteration # Stop iteration if index is not
             less than length
```

To make an instance iterable, we need to define the `__iter__` and `__next__` methods. `__iter__` method defines the start of the iterable. We start with the zero index to begin the sequence. `__next__` method defines the iteration pattern. If the index is less than the length of the sequence, we increment the index. Otherwise, we stop the iterable and end iteration.

The output is tested with the example:

```
print([n for n in GM])  # Output: [100, 110.0, 121.0, 133.1,
    146.41]
```

The output is as follows:

```
[100, 110.00, 121.00, 133.10, 146.41]
```

Which follows the expected output.

1.5 5

The task is to simulate a decay model. The class is identical except for the decay model. The formula for the decay model is:

$$\text{next value} = \text{current value} \times (1 - \text{rate})$$

```python
# Create class ExponentialDecayModel
class ExponentialDecayModel(BioModel):
    def __init__(self, start, rate):
        super().__init__(sequence=[])
        self.start = start # Class has input start
        self.rate = rate # Class has input rate

    # Callable method
    def __call__(self, length):
        self.sequence = [self.start] # Sequences begins with
            start parameter
        curr_val = self.start # Calculation begins with
            start parameter

        # Loop through and calculate sequence
        for _ in range(1, length):
            next_val = curr_val * (1 - self.rate) #
                Calculate next value
            self.sequence.append(next_val) # Add next value
                to sequence
            curr_val = next_val # Update current value to
                calculated value

        print(self.sequence) # Print final sequence

    # Define __len__ method to print length when called
    def __len__(self):
        return len(self.sequence)
```

Most of the `ExponentialDecayModel` subclass is the same as `ExponentialGrowthModel` . The only difference is the calculation in the `__call__` method. The formula is adjusted with the previously mentioned equation.

The output is tested with the example:

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

The output is as follows:

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

Which follows the expected output.

1.6 6

The task is to modify `BioModel` to allow for comparison between two instances using the `==` operator. If the sequences are the same length, they should be compared element-wise and return the count of matching elements. Otherwise raise a `ValueError`

```
    def __eq__(self, other):
        if not isinstance(other, BioModel):
            return NotImplemented
        if len(self.sequence) != len(other.sequence):
            raise ValueError("Two␣arrays␣are␣not␣equal␣in␣
                length!")

        count_match = 0
        for x, y in zip(self.sequence, other.sequence):
            if x == y:
                count_match += 1

        return count_match
```

The method `__eq__` defines what an instance should do when compared. When called, it compares to the "`other`" instance. We cannot compare the two if the other instance is not of type `BioModel`. Then, it compares the length for both sequences. If they are not equal, then an error `ValueError` is raised. If they are equal, each element will be compared pairwise. A counter `count_match` keeps track of the number of matching elements. Lastly, the value is returned once all values are compared.

The output is tested with the example:

```
print("Task␣6␣result")
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)  # Output: 1 (only the first element
    matches)

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

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

The output is as follows:

```
[100, 110.0, 121.0, 133.10, 146.41]
[100, 120.0, 144.0, 172.8, 207.36]
1
[100, 120.0, 144.0]
Traceback (most recent call last):
  File "c:\Users\Fanle\OneDrive\Desktop\Code\ECE60146\
      hw1_tasks.py", line 134, in <module>
    print(GM == GM3)  # Raises ValueError: Two arrays are
        not equal in length!
          ^^^^^^^^^
  File "c:\Users\Fanle\OneDrive\Desktop\Code\ECE60146\
      hw1_tasks.py", line 29, in __eq__
    raise ValueError("Two␣arrays␣are␣not␣equal␣in␣length!")
ValueError: Two arrays are not equal in length!
```

Which follows the expected output.

1.6.1 7

This task asks us to choose our `start` and `rate` parameter values and then show the results.

The values were chosen randomly by a random number generator. The `start` value for all tasks ranges from 100 to 999. The generator chose 285. Each `rate` is randomly selected from 0.1 to 0.9 except for the `GM` and `GM2` instances in example 6. They will be set to the same to show the count matches correctly.

The output is tested with the example:

```
# Task 3 result
print("Task␣3␣result")
GM = ExponentialGrowthModel(start=285, rate=0.9)
```

```
GM(length=5)
print(len(GM))  # Output: 5

# Task 4 Result
print("Task 4 result")
print([n for n in GM])

# Task 5 Result
DM = ExponentialDecayModel(start=285, rate=0.4)
DM(length=5)
print(len(DM))  # Output: 5
print([n for n in DM])

# Task 6 Result
GM = ExponentialGrowthModel(start=285, rate=0.3)
GM(length=5)

GM2 = ExponentialGrowthModel(start=285, rate=0.3)
GM2(length=5)

print(GM == GM2)  # Output: 5 (All output matches)

GM3 = ExponentialGrowthModel(start=285, rate=0.1)
GM3(length=3)  # [100, 120.0, 144.0]

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

The output is as follows:

```
Task 3 result
[285, 541.5, 1028.85, 1954.8149999999998,
    3714.1484999999993]
5
Task 4 result
[285, 541.5, 1028.85, 1954.8149999999998,
    3714.1484999999993]
Task 5 result
[285, 171.0, 102.6, 61.559999999999995, 36.93599999999999]
5
[285, 171.0, 102.6, 61.559999999999995, 36.93599999999999]
Task 6 result
[285, 370.5, 481.65000000000003, 626.1450000000001,
    813.9885000000002]
[285, 370.5, 481.65000000000003, 626.1450000000001,
    813.9885000000002]
5
[285, 313.5, 344.85]
Traceback (most recent call last):
  File "c:\Users\Fanle\OneDrive\Desktop\Code\ECE60146\
```

```
    hw1_choice_parameters.py", line 115, in <module>
    print(GM == GM3)  # Raises ValueError: Two arrays are not equal in length!
          ^^^^^^^^^^
  File "c:\Users\Fanle\OneDrive\Desktop\Code\ECE60146\hw1_choice_parameters.py", line 29, in __eq__
    raise ValueError("Two arrays are not equal in length!")
ValueError: Two arrays are not equal in length!
```

Which follows the expected output.

2 Bonus

2.1 1

This task asks us to create a new subclass called `CombinedBioModel`. It has the following requirements:

1. Generate a growth sequence using `ExponentialGrowthModel`
2. Generate a decay sequence using `ExponentialDecayModel`
3. Combine these sequences by multiplying the corresponding values from both sequences element-wise.

```python
# Create class CombinedBioModel
class CombinedBioModel(BioModel):
    def __init__(self, growth_start, growth_rate, decay_start, decay_rate):
        super().__init__(sequence = [])
        self.growth_model = ExponentialGrowthModel(start=growth_start, rate=growth_rate) # Class has growth model
        self.decay_model = ExponentialDecayModel(start=decay_start, rate=decay_rate) # Class has decay model

    # Callable method
    def __call__(self, length):
        # Initialize growth and decay model with length
        self.growth_model(length)
        self.decay_model(length)

        # Multiply elements pairwise from growth and decay model
```

```
        self.sequence = [g * d for g, d in zip(self.
            growth_model.sequence, self.decay_model.sequence)
            ]

        # Print calculated sequence
        print(self.sequence)
```

The class `CombinedBioModel` initializes a growth model and decay model instances using the input parameters. The `__call__` method receives an input parameter `length`, initializing the two instances. The pairwise product is then calculated using list comprehension.

The output is tested with the example:

```
print("Bonus 1 result")
CBM = CombinedBioModel(growth_start=100, growth_rate=0.1,
    decay_start=1.0, decay_rate=0.05)
CBM(length=5)  # Output: [100.0, 104.5, 109.2, 114.12,
    119.25]
```

The output is as follows:

```
[100, 110.00, 121.00, 133.10, 146.41]
[1.0, 0.95, 0.9025, 0.86, 0.81]
[100.0, 104.50, 109.2, 114.12, 119.25]
```

Which follows the expected output.

2.2 2

This task asks us to visualize the comparison between two growth and decay rates.

```
# Two growth and decay rates
# Length is set to 10 for visualization
# Growth and decay set at same start point to visualize
    difference
growth_rate_1 = 0.1
growth_rate_2 = 0.5
decay_rate_1 = 0.02
decay_rate_2 = 0.1
length = 10
growth_start = 10
decay_start = 10

# Combination of Growth Rates
print("Growth rate 1, Decay rate 1")
```

```
combined_model_1 = CombinedBioModel(growth_start=
    growth_start, growth_rate=growth_rate_1, decay_start=
    decay_start, decay_rate=decay_rate_1)
combined_model_1(length)
growth_model_1 = combined_model_1.growth_model
decay_model_1 = combined_model_1.decay_model

print("Growth rate 2, Decay rate 2")
combined_model_2 = CombinedBioModel(growth_start=
    growth_start, growth_rate=growth_rate_2, decay_start=
    decay_start, decay_rate=decay_rate_2)
combined_model_2(length)
growth_model_2 = combined_model_2.growth_model
decay_model_2 = combined_model_2.decay_model

print("Growth rate 2, Decay rate 1")
combined_model_3 = CombinedBioModel(growth_start=
    growth_start, growth_rate=growth_rate_2, decay_start=
    decay_start, decay_rate=decay_rate_1)
combined_model_3(length)

print("Growth rate 1, Decay rate 2")
combined_model_4 = CombinedBioModel(growth_start=
    growth_start, growth_rate=growth_rate_1, decay_start=
    decay_start, decay_rate=decay_rate_2)
combined_model_4(length)

fig, axs = plt.subplots(2, 2, figsize=(12, 10))
fig.suptitle("Growth, Decay, and Combined Dynamics",
    fontsize=16)

# Plot subplots
axs[0, 0].plot(range(length), growth_model_1.sequence, label
    ="Growth", color="blue", linestyle="solid")
axs[0, 0].plot(range(length), decay_model_1.sequence, label=
    "Decay", color="red", linestyle="dashed")
axs[0, 0].plot(range(length), combined_model_1.sequence,
    label="Combined", color="green", linestyle="dotted")
axs[0, 0].set_title(f"Growth Rate: {growth_rate_1}, Decay
    Rate: {decay_rate_1}")
axs[0, 0].set_xlabel("Iteration")
axs[0, 0].set_ylabel("Value")
axs[0, 0].legend()

axs[0, 1].plot(range(length), growth_model_2.sequence, label
    ="Growth", color="blue", linestyle="solid")
axs[0, 1].plot(range(length), decay_model_1.sequence, label=
    "Decay", color="red", linestyle="dashed")
axs[0, 1].plot(range(length), combined_model_3.sequence,
    label="Combined", color="green", linestyle="dotted")
```

```
axs[0, 1].set_title(f"Growth Rate: {growth_rate_2}, Decay Rate: {decay_rate_1}")
axs[0, 1].set_xlabel("Iteration")
axs[0, 1].set_ylabel("Value")
axs[0, 1].legend()

axs[1, 0].plot(range(length), growth_model_1.sequence, label="Growth", color="blue", linestyle="solid")
axs[1, 0].plot(range(length), decay_model_2.sequence, label="Decay", color="red", linestyle="dashed")
axs[1, 0].plot(range(length), combined_model_4.sequence, label="Combined", color="green", linestyle="dotted")
axs[1, 0].set_title(f"Growth Rate: {growth_rate_1}, Decay Rate: {decay_rate_2}")
axs[1, 0].set_xlabel("Iteration")
axs[1, 0].set_ylabel("Value")
axs[1, 0].legend()

axs[1, 1].plot(range(length), growth_model_2.sequence, label="Growth", color="blue", linestyle="solid")
axs[1, 1].plot(range(length), decay_model_2.sequence, label="Decay", color="red", linestyle="dashed")
axs[1, 1].plot(range(length), combined_model_2.sequence, label="Combined", color="green", linestyle="dotted")
axs[1, 1].set_title(f"Growth Rate: {growth_rate_2}, Decay Rate: {decay_rate_2}")
axs[1, 1].set_xlabel("Iteration")
axs[1, 1].set_ylabel("Value")
axs[1, 1].legend()

# Set all y axis equal
for ax in axs.flat:
    ax.set_ylim(0, 200)

plt.show()
```

Values were chosen at random. The plots are shown in Figure 1.

3 Source Code

The code for all the tasks and bonus is presented below

```
hw1_tasks.py

import matplotlib.pyplot as plt

# Task 1
```

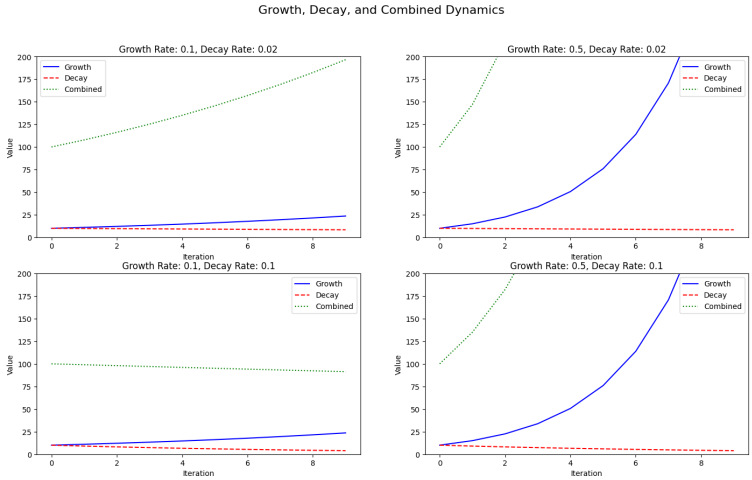

Figure 1: Generated growth, decay and combined sequences for two growth and two decay rates

```python
# Create class BioModel
class BioModel(object):
    def __init__(self, sequence):
        self.sequence = sequence # Class has input sequence

    #Task 4
    # Define iter method to start iterable
    def __iter__(self):
        self.index = 0 # Start index at 0
        return self

    # Define next method to continue iterable
    def __next__(self):
        # Logic to determine when to stop iteration
        if self.index < len(self.sequence):
            self.index += 1
            return self.sequence[self.index - 1] # Too lazy to add more lines. Subtract one from index for accurate output
        else:
            raise StopIteration # Stop iteration if index is not less than length

    # Task 6
    def __eq__(self, other):
        if not isinstance(other, BioModel):
            return NotImplemented
```

```
30          if len(self.sequence) != len(other.sequence):
31              raise ValueError("Two arrays are not equal in
                    length!")
32
33          count_match = 0
34          for x, y in zip(self.sequence, other.sequence):
35              if x == y:
36                  count_match += 1
37
38          return count_match
39
40  # Task 2
41  # Create class ExponentialGrowthModel
42  class ExponentialGrowthModel(BioModel):
43      def __init__(self, start, rate):
44          super().__init__(sequence=[])
45          self.start = start # Class has input start
46          self.rate = rate # Class has input rate
47
48      # Task 3
49      # Callable method
50      def __call__(self, length):
51          self.sequence = [self.start] # Sequences begins with
                start parameter
52          curr_val = self.start # Calculation begins with
                start parameter
53
54          # Loop through and calculate sequence
55          for _ in range(1, length):
56              next_val = curr_val * (1 + self.rate) #
                    Calculate next value
57              self.sequence.append(next_val) # Add next value
                    to sequence
58              curr_val = next_val # Update current value to
                    calculated value
59
60          print(self.sequence) # Print final sequence
61
62      # Define __len__ method to print length when called
63      def __len__(self):
64          return len(self.sequence)
65
66  # Task 5
67  # Create class ExponentialDecayModel
68  class ExponentialDecayModel(BioModel):
69      def __init__(self, start, rate):
70          super().__init__(sequence=[])
71          self.start = start # Class has input start
72          self.rate = rate # Class has input rate
73
```

```
74      # Callable method
75      def __call__(self, length):
76          self.sequence = [self.start] # Sequences begins with
                start parameter
77          curr_val = self.start # Calculation begins with
                start parameter
78
79          # Loop through and calculate sequence
80          for _ in range(1, length):
81              next_val = curr_val * (1 - self.rate) #
                    Calculate next value
82              self.sequence.append(next_val) # Add next value
                    to sequence
83              curr_val = next_val # Update current value to
                    calculated value
84
85          print(self.sequence) # Print final sequence
86
87      # Define __len__ method to print length when called
88      def __len__(self):
89          return len(self.sequence)
90
91  # Bonus 1
92  # Create class CombinedBioModel
93  class CombinedBioModel(BioModel):
94      def __init__(self, growth_start, growth_rate,
            decay_start, decay_rate):
95          super().__init__(sequence = [])
96          self.growth_model = ExponentialGrowthModel(start=
                growth_start, rate=growth_rate) # Class has
                growth model
97          self.decay_model = ExponentialDecayModel(start=
                decay_start, rate=decay_rate) # Class has decay
                model
98
99      # Callable method
100     def __call__(self, length):
101         # Initialize growth and decay model with length
102         self.growth_model(length)
103         self.decay_model(length)
104
105         # Multiply elements pairwise from growth and decay
                model
106         self.sequence = [g * d for g, d in zip(self.
                growth_model.sequence, self.decay_model.sequence)
                ]
107
108         # Print calculated sequence
109         print(self.sequence)
110
```

```python
# Task 3 result
print("Task 3 result")
GM = ExponentialGrowthModel(start=100, rate=0.1)
GM(length=5)  # Output: [100, 110.0, 121.0, 133.1, 146.41]
print(len(GM))  # Output: 5

# Task 4 Result
print("Task 4 result")
print([n for n in GM])  # Output: [100, 110.0, 121.0, 133.1,
    146.41]

# Task 5 Result
print("Task 5 result")
DM = ExponentialDecayModel(start=100, rate=0.2)
DM(length=5)  # Output: [100, 80.0, 64.0, 51.2, 40.96]
print(len(DM))  # Output: 5
print([n for n in DM])  # Output: [100, 80.0, 64.0, 51.2,
    40.96]

# Task 6 Result
print("Task 6 result")
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)  # Output: 1 (only the first element
    matches)

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

# Commented out for running bonus outputs
# print(GM == GM3)  # Raises ValueError: Two arrays are not
    equal in length!

# Bonus 1
print("Bonus 1 result")
CBM = CombinedBioModel(growth_start=100, growth_rate=0.1,
    decay_start=1.0, decay_rate=0.05)
CBM(length=5)  # Output: [100.0, 104.5, 109.2, 114.12,
    119.25]

# Bonus 2
print("Bonus 2 result")

# Two growth and decay rates
# Length is set to 10 for visualization
```

```
# Growth and decay set at same start point to visualize difference
growth_rate_1 = 0.1
growth_rate_2 = 0.5
decay_rate_1 = 0.02
decay_rate_2 = 0.1
length = 10
growth_start = 10
decay_start = 10

# Combination of Growth Rates
print("Growth rate 1, Decay rate 1")
combined_model_1 = CombinedBioModel(growth_start=growth_start, growth_rate=growth_rate_1, decay_start=decay_start, decay_rate=decay_rate_1)
combined_model_1(length)
growth_model_1 = combined_model_1.growth_model
decay_model_1 = combined_model_1.decay_model

print("Growth rate 2, Decay rate 2")
combined_model_2 = CombinedBioModel(growth_start=growth_start, growth_rate=growth_rate_2, decay_start=decay_start, decay_rate=decay_rate_2)
combined_model_2(length)
growth_model_2 = combined_model_2.growth_model
decay_model_2 = combined_model_2.decay_model

print("Growth rate 2, Decay rate 1")
combined_model_3 = CombinedBioModel(growth_start=growth_start, growth_rate=growth_rate_2, decay_start=decay_start, decay_rate=decay_rate_1)
combined_model_3(length)

print("Growth rate 1, Decay rate 2")
combined_model_4 = CombinedBioModel(growth_start=growth_start, growth_rate=growth_rate_1, decay_start=decay_start, decay_rate=decay_rate_2)
combined_model_4(length)

fig, axs = plt.subplots(2, 2, figsize=(12, 10))
fig.suptitle("Growth, Decay, and Combined Dynamics", fontsize=16)

# Plot subplots
axs[0, 0].plot(range(length), growth_model_1.sequence, label="Growth", color="blue", linestyle="solid")
axs[0, 0].plot(range(length), decay_model_1.sequence, label="Decay", color="red", linestyle="dashed")
axs[0, 0].plot(range(length), combined_model_1.sequence, label="Combined", color="green", linestyle="dotted")
```

```
axs[0, 0].set_title(f"Growth Rate: {growth_rate_1}, Decay Rate: {decay_rate_1}")
axs[0, 0].set_xlabel("Iteration")
axs[0, 0].set_ylabel("Value")
axs[0, 0].legend()

axs[0, 1].plot(range(length), growth_model_2.sequence, label="Growth", color="blue", linestyle="solid")
axs[0, 1].plot(range(length), decay_model_1.sequence, label="Decay", color="red", linestyle="dashed")
axs[0, 1].plot(range(length), combined_model_3.sequence, label="Combined", color="green", linestyle="dotted")
axs[0, 1].set_title(f"Growth Rate: {growth_rate_2}, Decay Rate: {decay_rate_1}")
axs[0, 1].set_xlabel("Iteration")
axs[0, 1].set_ylabel("Value")
axs[0, 1].legend()

axs[1, 0].plot(range(length), growth_model_1.sequence, label="Growth", color="blue", linestyle="solid")
axs[1, 0].plot(range(length), decay_model_2.sequence, label="Decay", color="red", linestyle="dashed")
axs[1, 0].plot(range(length), combined_model_4.sequence, label="Combined", color="green", linestyle="dotted")
axs[1, 0].set_title(f"Growth Rate: {growth_rate_1}, Decay Rate: {decay_rate_2}")
axs[1, 0].set_xlabel("Iteration")
axs[1, 0].set_ylabel("Value")
axs[1, 0].legend()

axs[1, 1].plot(range(length), growth_model_2.sequence, label="Growth", color="blue", linestyle="solid")
axs[1, 1].plot(range(length), decay_model_2.sequence, label="Decay", color="red", linestyle="dashed")
axs[1, 1].plot(range(length), combined_model_2.sequence, label="Combined", color="green", linestyle="dotted")
axs[1, 1].set_title(f"Growth Rate: {growth_rate_2}, Decay Rate: {decay_rate_2}")
axs[1, 1].set_xlabel("Iteration")
axs[1, 1].set_ylabel("Value")
axs[1, 1].legend()

# Set all y axis equal
for ax in axs.flat:
    ax.set_ylim(0, 200)

plt.show()
```

hw1_choice_parameters.py

```
3  import matplotlib.pyplot as plt
4
5  # Task 1
6  # Create class BioModel
7  class BioModel(object):
8      def __init__(self, sequence):
9          self.sequence = sequence # Class has input sequence
10
11     #Task 4
12     # Define iter method to start iterable
13     def __iter__(self):
14         self.index = 0 # Start index at 0
15         return self
16
17     # Define next method to continue iterable
18     def __next__(self):
19         # Logic to determine when to stop iteration
20         if self.index < len(self.sequence):
21             self.index += 1
22             return self.sequence[self.index - 1] # Too lazy
                   to add more lines. Subtract one from index
                   for accurate output
23         else:
24             raise StopIteration # Stop iteration if index is
                    not less than length
25
26     # Task 6
27     def __eq__(self, other):
28         if not isinstance(other, BioModel):
29             return NotImplemented
30         if len(self.sequence) != len(other.sequence):
31             raise ValueError("Two␣arrays␣are␣not␣equal␣in␣
                   length!")
32
33         count_match = 0
34         for x, y in zip(self.sequence, other.sequence):
35             if x == y:
36                 count_match += 1
37
38         return count_match
39
40 # Task 2
41 # Create class ExponentialGrowthModel
42 class ExponentialGrowthModel(BioModel):
43     def __init__(self, start, rate):
44         super().__init__(sequence=[])
45         self.start = start # Class has input start
46         self.rate = rate # Class has input rate
47
48     # Task 3
```

```
    # Callable method
    def __call__(self, length):
        self.sequence = [self.start] # Sequences begins with
            start parameter
        curr_val = self.start # Calculation begins with
            start parameter

        # Loop through and calculate sequence
        for _ in range(1, length):
            next_val = curr_val * (1 + self.rate) #
                Calculate next value
            self.sequence.append(next_val) # Add next value
                to sequence
            curr_val = next_val # Update current value to
                calculated value

        print(self.sequence) # Print final sequence

    # Define __len__ method to print length when called
    def __len__(self):
        return len(self.sequence)

# Task 5
class ExponentialDecayModel(BioModel):
    def __init__(self, start, rate):
        super().__init__(sequence=[])
        self.start = start
        self.rate = rate

    def __call__(self, length):
        self.sequence = [self.start]
        curr_val = self.start

        for _ in range(1, length):
            next_val = curr_val * (1 - self.rate)
            self.sequence.append(next_val)
            curr_val = next_val

        print(self.sequence)

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

# Task 3 result
print("Task␣3␣result")
GM = ExponentialGrowthModel(start=285, rate=0.9)
GM(length=5)
print(len(GM))  # Output: 5

# Task 4 Result
```

```
print("Task␣4␣result")
print([n for n in GM])

# Task 5 Result
print("Task␣5␣result")
DM = ExponentialDecayModel(start=285, rate=0.4)
DM(length=5)
print(len(DM))  # Output: 5
print([n for n in DM])

# Task 6 Result
print("Task␣6␣result")
GM = ExponentialGrowthModel(start=285, rate=0.3)
GM(length=5)

GM2 = ExponentialGrowthModel(start=285, rate=0.3)
GM2(length=5)

print(GM == GM2)  # Output: 5 (All output matches)

GM3 = ExponentialGrowthModel(start=285, rate=0.1)
GM3(length=3)  # [100, 120.0, 144.0]

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