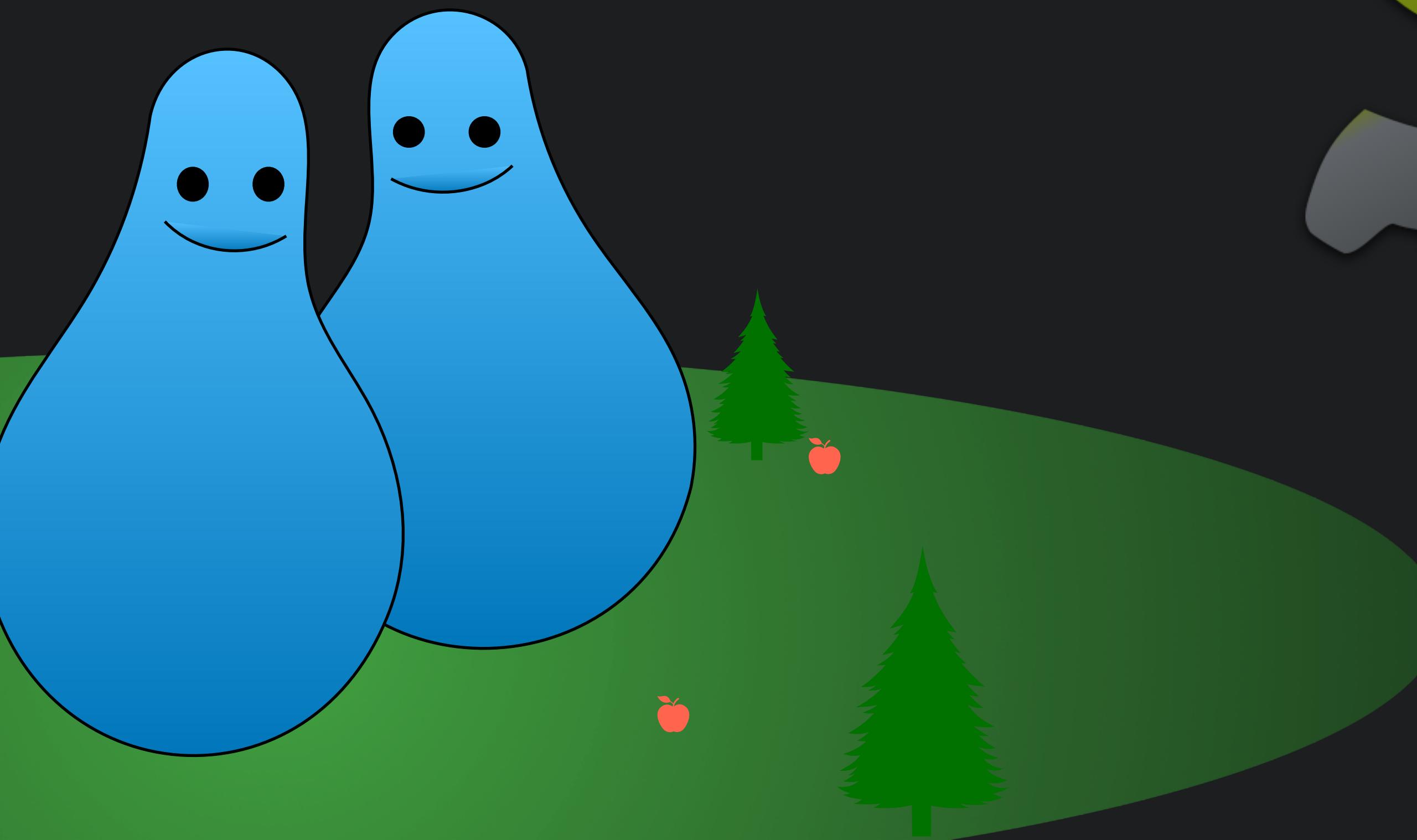
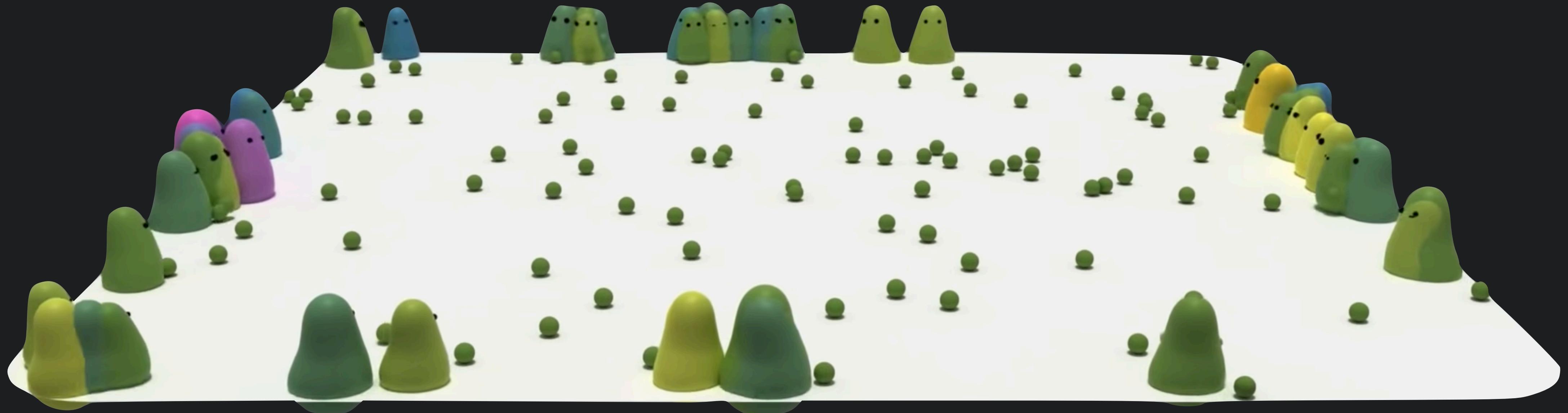


A Biological Simulation in *Python* using

The logo for SIMPY, featuring the word "SIMPY" in a stylized font where the letters S, I, m, P, and Y are colored green, while the letter y is grey. To the left of the text is a green Python icon, which is a green snake-like shape coiled around a dark grey gear.

Stefano Mangini, Unipv, July 2020

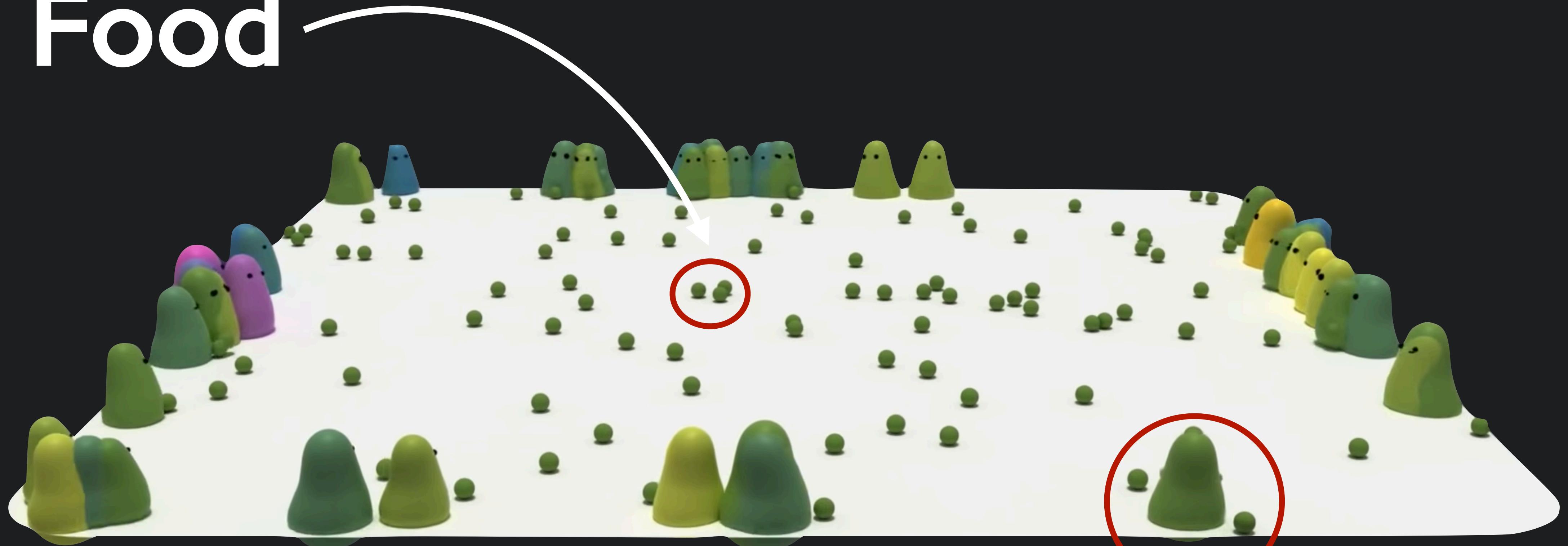
PRIMER



Link Youtube: [Primer](#)

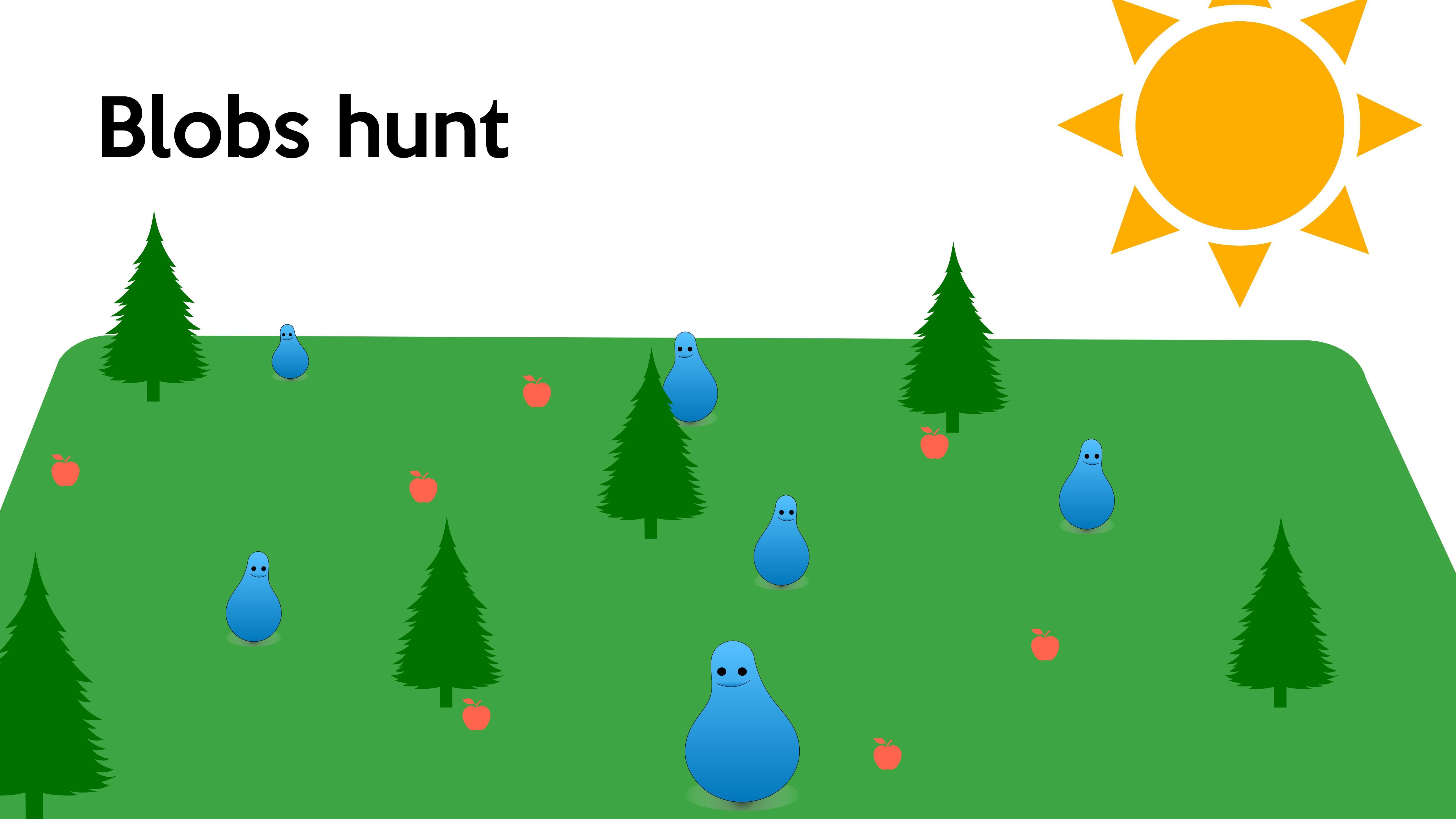
Forest

Food



Blob

Blobs hunt



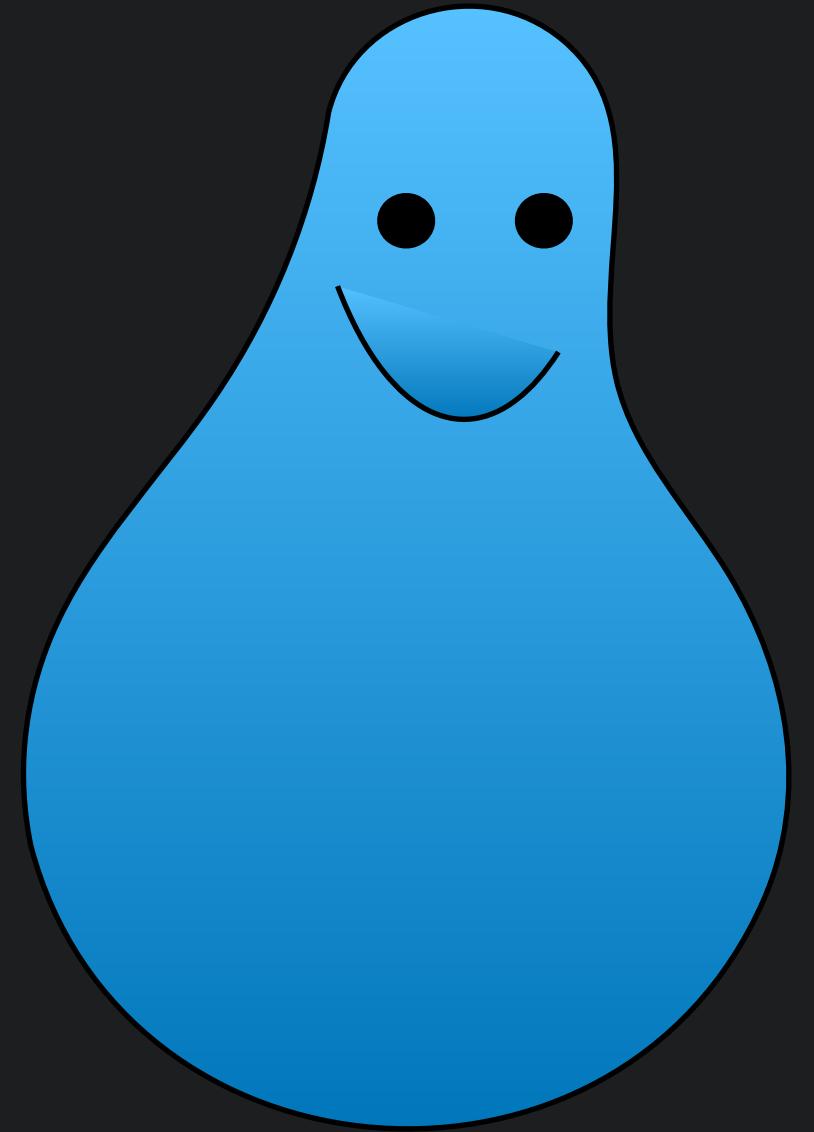
Blobs sleep

Food regrows back

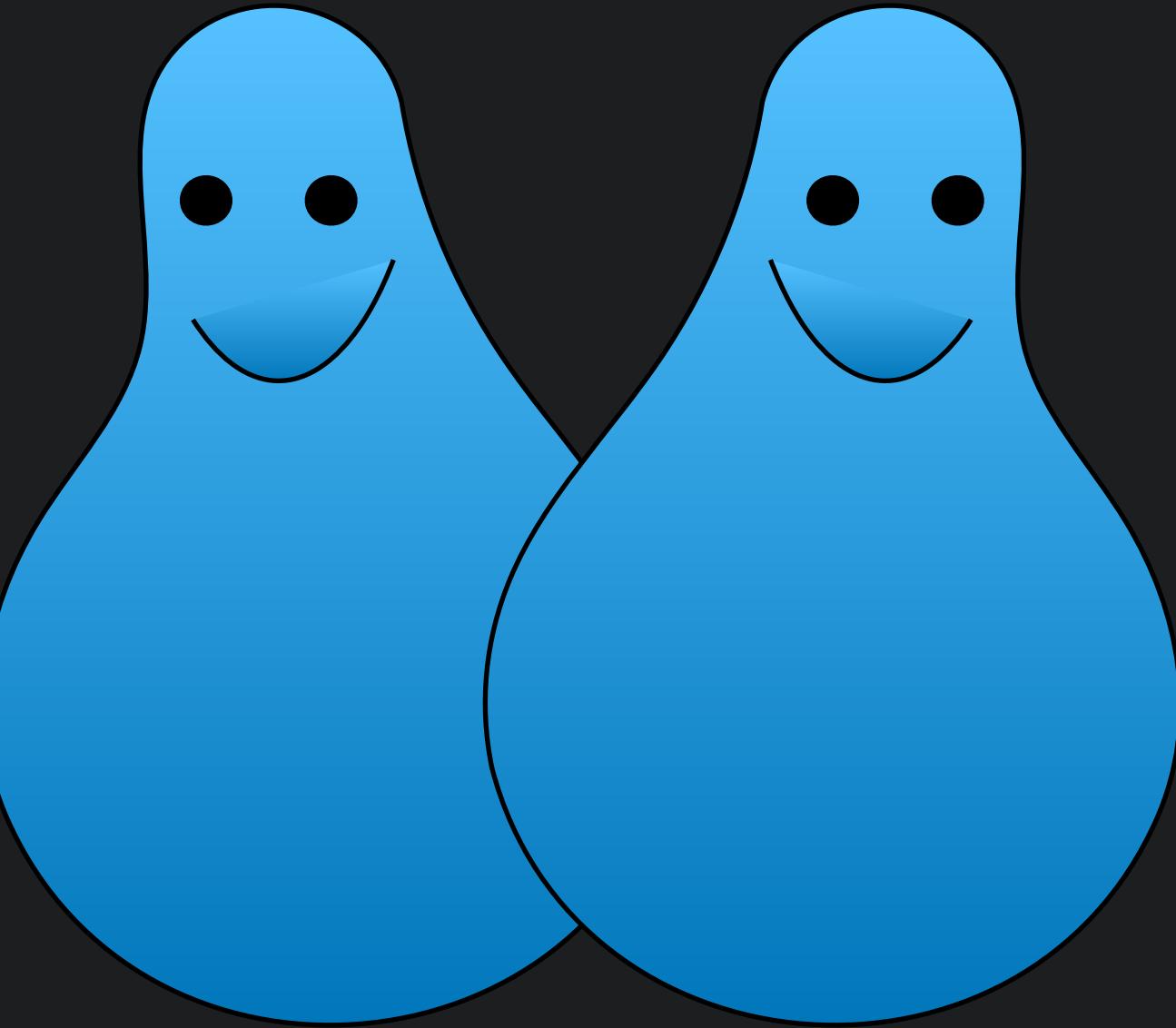




If food_eaten < 1



If food_eaten > 3





simPy?



The logo consists of the word "SimPy" in a bold, sans-serif font. The letter "S" is stylized to look like a green and yellow snake with a small eye. The letters "imPy" are in a dark gray color. A question mark symbol is integrated into the "P" character.

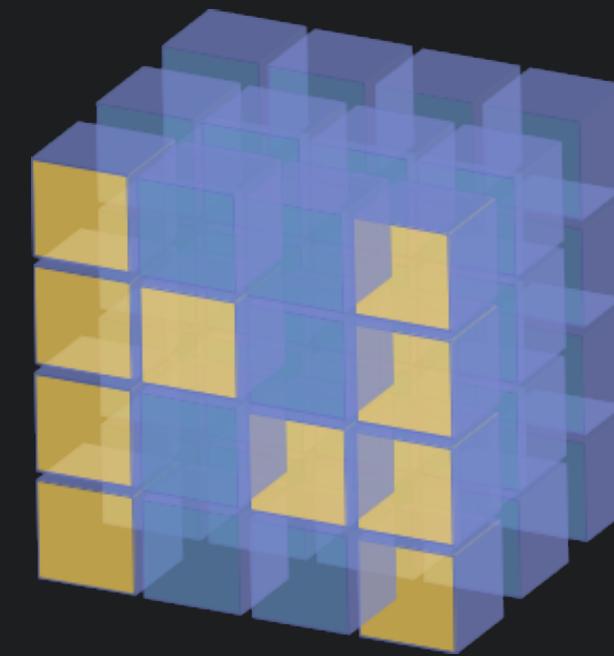
SimPy?



The logo consists of the word "SimPy" in a bold, sans-serif font. The letter "S" is stylized to look like a green and yellow snake with a small eye. The letters "imPy" are in a dark gray color. An exclamation mark symbol is integrated into the "y" character.

SimPy!

Required packages:



NumPy



tqdm



Scipy

Forest

```
class Forest:
    """
    The Forest is the natural environment where the Blobs live.
    The each day in the simulation is divided in two: Daylight and Nighttime.
    The Forest starts with a limited food supply that the Blobs eat during Daylight in order to survive.
    At nighttime, the blobs goes to sleep and the Forest grows new food.

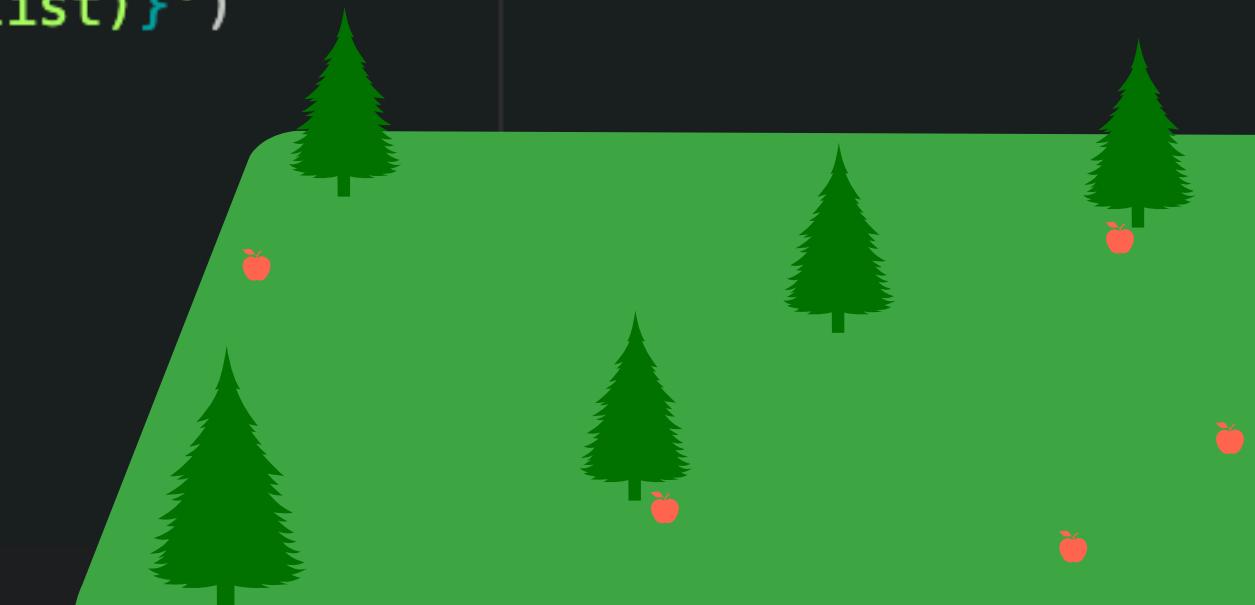
    """

    def __init__(self, env):
        self.food = simpy.Container(env, init=INITIAL_FOOD)
        self.time_proc = env.process(self.day_cycle(env))

    # Define the day to night cycle, and prints when they occur
    def day_cycle(self, env):
        while True:
            day = int(env.now/DAY)
            print(f'\nSTART OF DAY {day} (sim. time {env.now})')
            print(f' > Forest food level: {self.food.level}')
            print(f' > Living Blobs: {count_alive_blobs(Blob_list)}')
            living_blobs.append(count_alive_blobs(Blob_list))
            food_level_list.append(self.food.level)
            yield env.timeout(DAYLIGHT) # Wait for the sun to set.

            print(f'\nNIGHTTIME STARTS at {env.now}')
            # living_blobs.append(count_alive_blobs(Blob_list))
            yield env.process(self.food_growth(env)) # Food grows
            yield env.timeout(NIGHTTIME) # Wait for the sun to rise
            print(f' > Survived Blobs: {count_alive_blobs(Blob_list)}')

    # New food is added to the Forest
    def food_growth(self, env):
        print(f' > Forest food level: {self.food.level}')
        new_food = FOOD_RATE_PRODUCTION
        yield self.food.put(new_food)
```



Forest

Simpy Container and start of processes

```
class Forest:
    """
    The Forest is the natural environment where the Blobs live.
    The each day in the simulation is divided in two: Daylight and Nighttime.
    The Forest starts with a limited food supply that the Blobs eat during Daylight in order to survive.
    At nighttime, the blobs goes to sleep and the Forest grows new food.

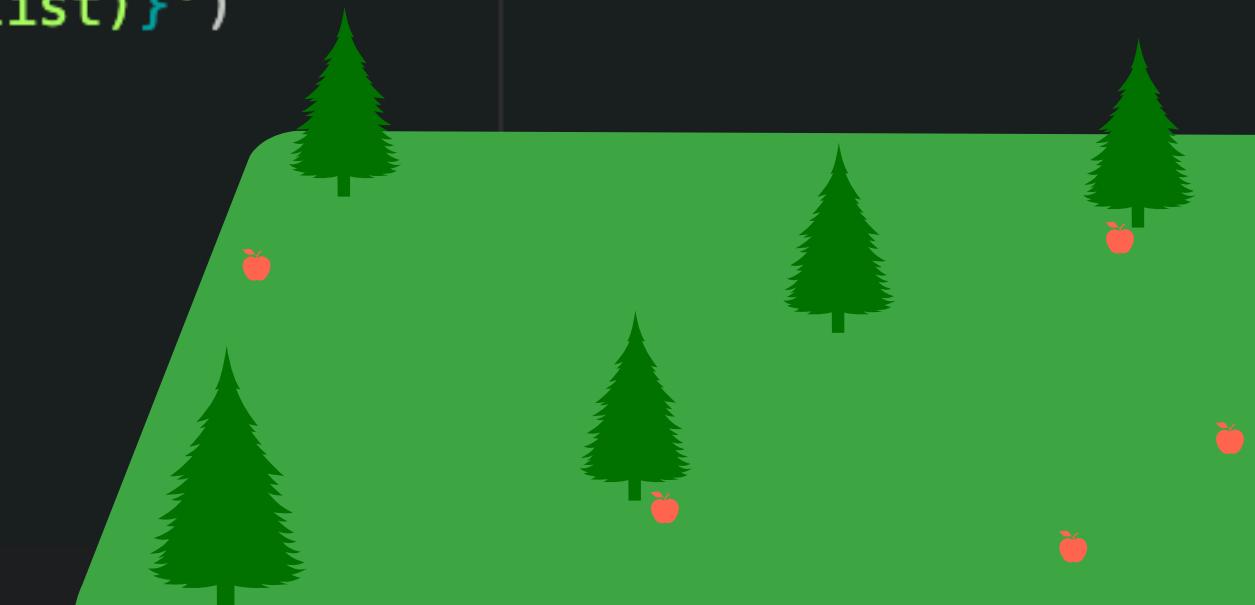
    """

    def __init__(self, env):
        self.food = simpy.Container(env, init=INITIAL_FOOD)
        self.time_proc = env.process(self.day_cycle(env))

    # Define the day to night cycle, and prints when they occur
    def day_cycle(self, env):
        while True:
            day = int(env.now/DAY)
            print(f'\nSTART OF DAY {day} (sim. time {env.now})')
            print(f' > Forest food level: {self.food.level}')
            print(f' > Living Blobs: {count_alive_blobs(Blob_list)}')
            living_blobs.append(count_alive_blobs(Blob_list))
            food_level_list.append(self.food.level)
            yield env.timeout(DAYLIGHT) # Wait for the sun to set.

            print(f'\nNIGHTTIME STARTS at {env.now}')
            # living_blobs.append(count_alive_blobs(Blob_list))
            yield env.process(self.food_growth(env)) # Food grows
            yield env.timeout(NIGHTTIME) # Wait for the sun to rise
            print(f' > Survived Blobs: {count_alive_blobs(Blob_list)}')

    # New food is added to the Forest
    def food_growth(self, env):
        print(f' > Forest food level: {self.food.level}')
        new_food = FOOD_RATE_PRODUCTION
        yield self.food.put(new_food)
```

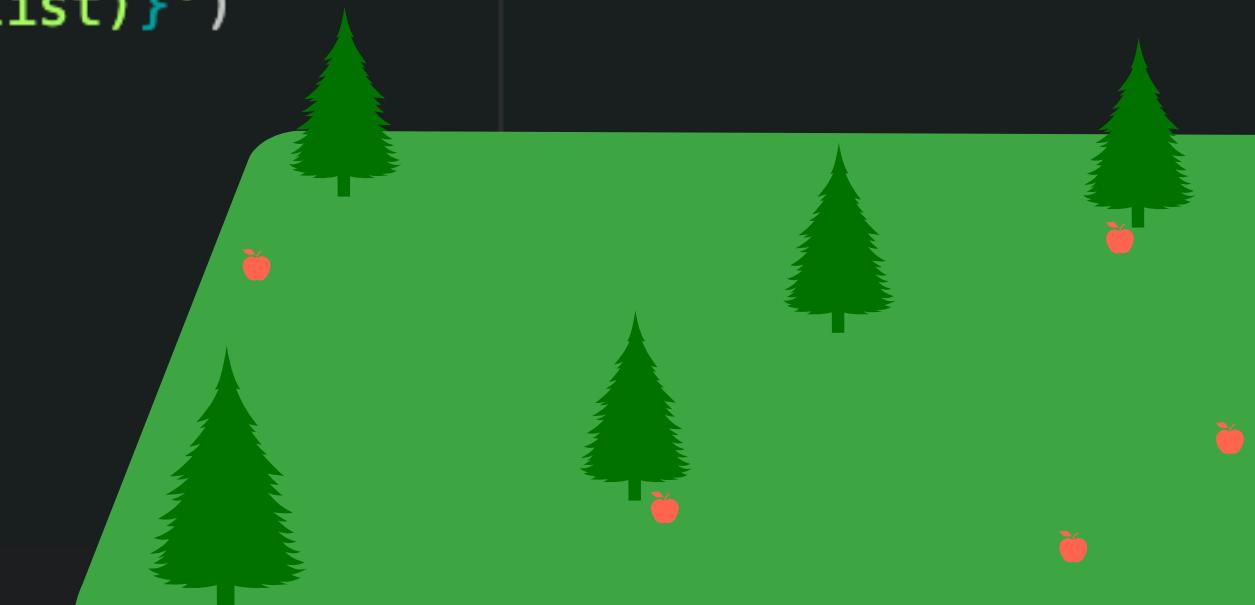


Forest

Simpy Container and start of processes

Time passes

```
class Forest:  
    """  
        The Forest is the natural environment where the Blobs live.  
        The each day in the simulation is divided in two: Daylight and Nighttime.  
        The Forest starts with a limited food supply that the Blobs eat during Daylight in order to survive.  
        At nighttime, the blobs goes to sleep and the Forest grows new food.  
    """  
  
    def __init__(self, env):  
        self.food = simpy.Container(env, init=INITIAL_FOOD)  
        self.time_proc = env.process(self.day_cycle(env))  
  
    # Define the day to night cycle, and prints when they occur  
    def day_cycle(self, env):  
        while True:  
            day = int(env.now/DAY)  
            print(f'\nSTART OF DAY {day} (sim. time {env.now})')  
            print(f' > Forest food level: {self.food.level}')  
            print(f' > Living Blobs: {count_alive_blobs(Blob_list)}')  
            living_blobs.append(count_alive_blobs(Blob_list))  
            food_level_list.append(self.food.level)  
            yield env.timeout(DAYLIGHT) # Wait for the sun to set.  
  
            print(f'\nNIGHTTIME STARTS at {env.now}')  
            # living_blobs.append(count_alive_blobs(Blob_list))  
            yield env.process(self.food_growth(env)) # Food grows  
            yield env.timeout(NIGHTTIME) # Wait for the sun to rise  
            print(f' > Survived Blobs: {count_alive_blobs(Blob_list)}')  
  
    # New food is added to the Forest  
    def food_growth(self, env):  
        print(f' > Forest food level: {self.food.level}')  
        new_food = FOOD_RATE_PRODUCTION  
        yield self.food.put(new_food)
```



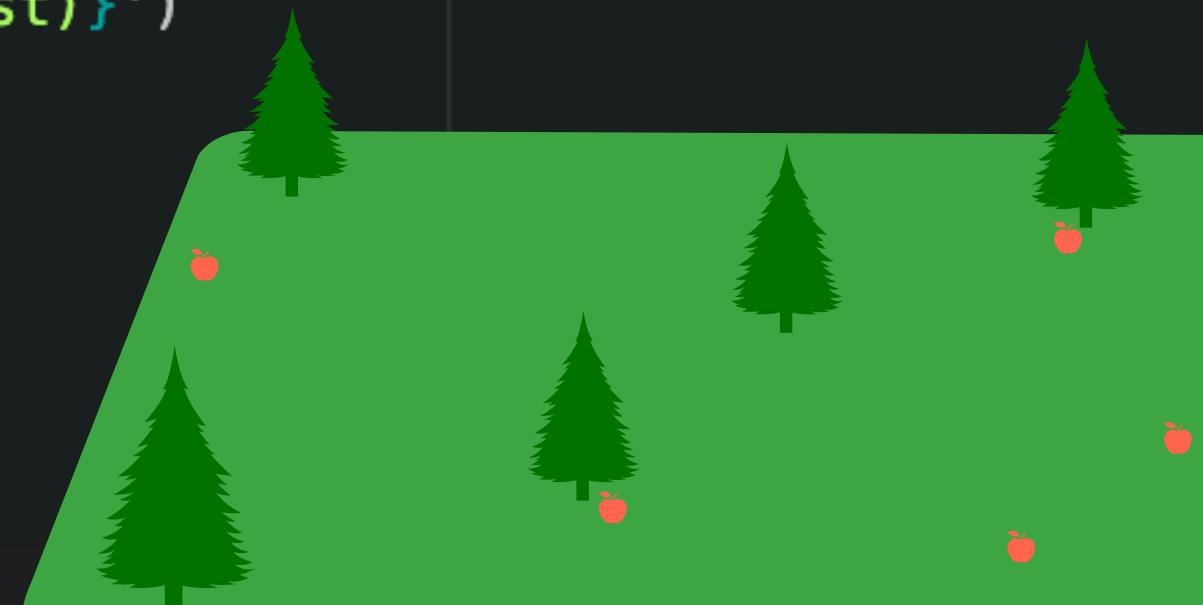
Forest

Simpy Container and start of processes

Time passes

Food grows
(with .put method)

```
class Forest:  
    """  
        The Forest is the natural environment where the Blobs live.  
        The each day in the simulation is divided in two: Daylight and Nighttime.  
        The Forest starts with a limited food supply that the Blobs eat during Daylight in order to survive.  
        At nighttime, the blobs goes to sleep and the Forest grows new food.  
  
    """  
  
    def __init__(self, env):  
        self.food = simpy.Container(env, init=INITIAL_FOOD)  
        self.time_proc = env.process(self.day_cycle(env))  
  
    # Define the day to night cycle, and prints when they occur  
    def day_cycle(self, env):  
        while True:  
            day = int(env.now/DAY)  
            print(f'\nSTART OF DAY {day} (sim. time {env.now})')  
            print(f' > Forest food level: {self.food.level}')  
            print(f' > Living Blobs: {count_alive_blobs(Blob_list)}')  
            living_blobs.append(count_alive_blobs(Blob_list))  
            food_level_list.append(self.food.level)  
            yield env.timeout(DAYLIGHT) # Wait for the sun to set.  
  
            print(f'\nNIGHTTIME STARTS at {env.now}')  
            # living_blobs.append(count_alive_blobs(Blob_list))  
            yield env.process(self.food_growth(env)) # Food grows  
            yield env.timeout(NIGHTTIME) # Wait for the sun to rise  
            print(f' > Survived Blobs: {count_alive_blobs(Blob_list)}')  
  
    # New food is added to the Forest  
    def food_growth(self, env):  
        print(f' > Forest food level: {self.food.level}')  
        new_food = FOOD_RATE_PRODUCTION  
        yield self.food.put(new_food)
```

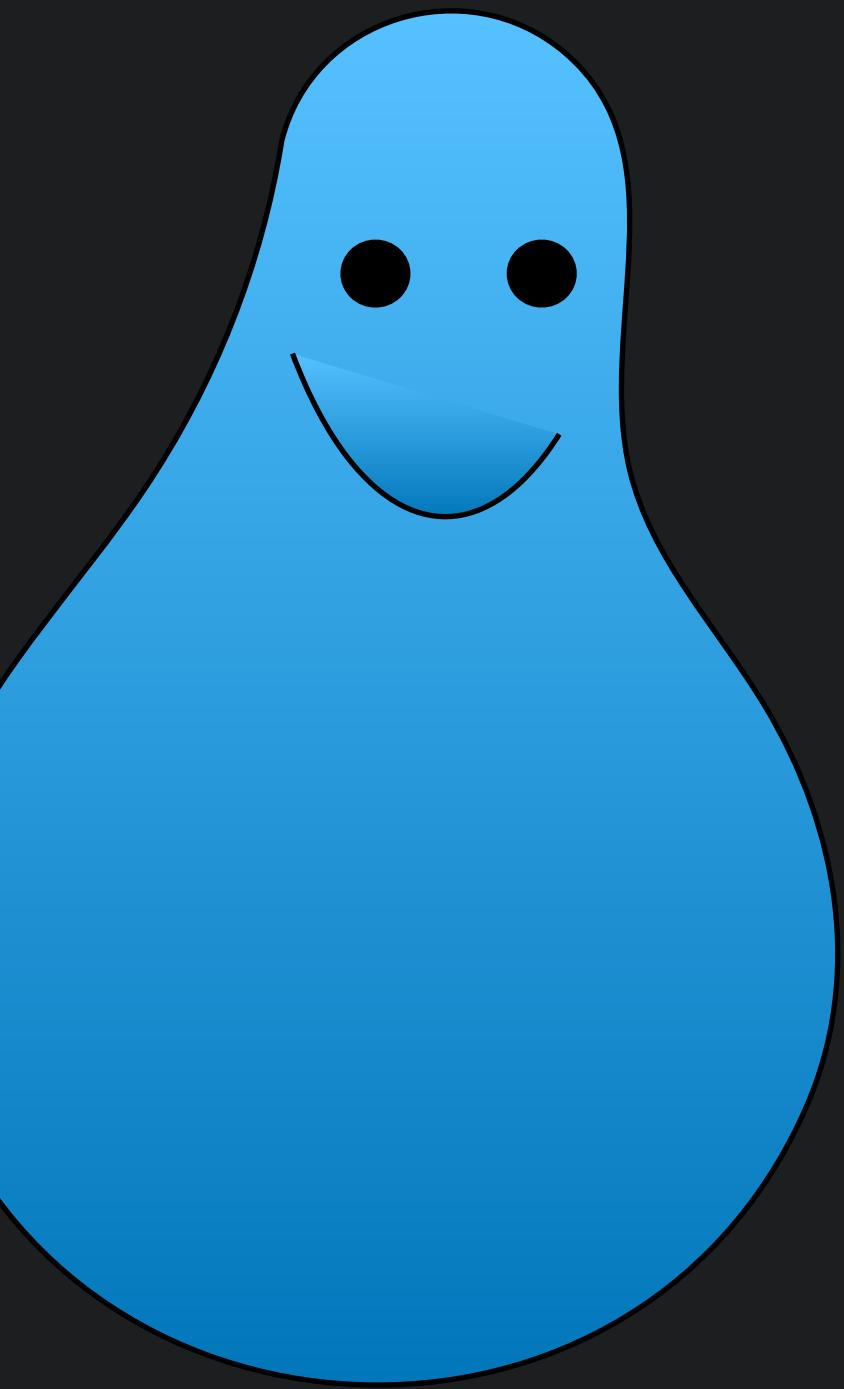


Blob

```
class Blob:  
    """  
        A Blob is a living creature populating the Forest.  
        Each blob can be either alive or dead, depending on its ability to search for food.  
        If a Blob eats enough food during a day, it survives to the next day. Otherwise, it dies.  
    """
```

The Blob only search for food during Daylight. When the nighttime arrives, it goes to sleep.

```
"""  
def __init__(self, env, forest, name):  
    self.env = env  
    self.name = name # Name of the blob  
  
    self.alive = True # Denotes the Blob's state: alive or dead  
    self.isHunting = True # Denotes if the Blob is searching for food  
    self.food_requirements = 1 # Minimum food to eat per day to survive  
    self.food_eaten = 0 # Amount of food eaten during the current day  
    self.can_reproduce = False  
  
    self.sleep_proc = env.process(self.sleep(env, forest))  
    self.hunt_proc = env.process(self.hunt(env, forest))
```



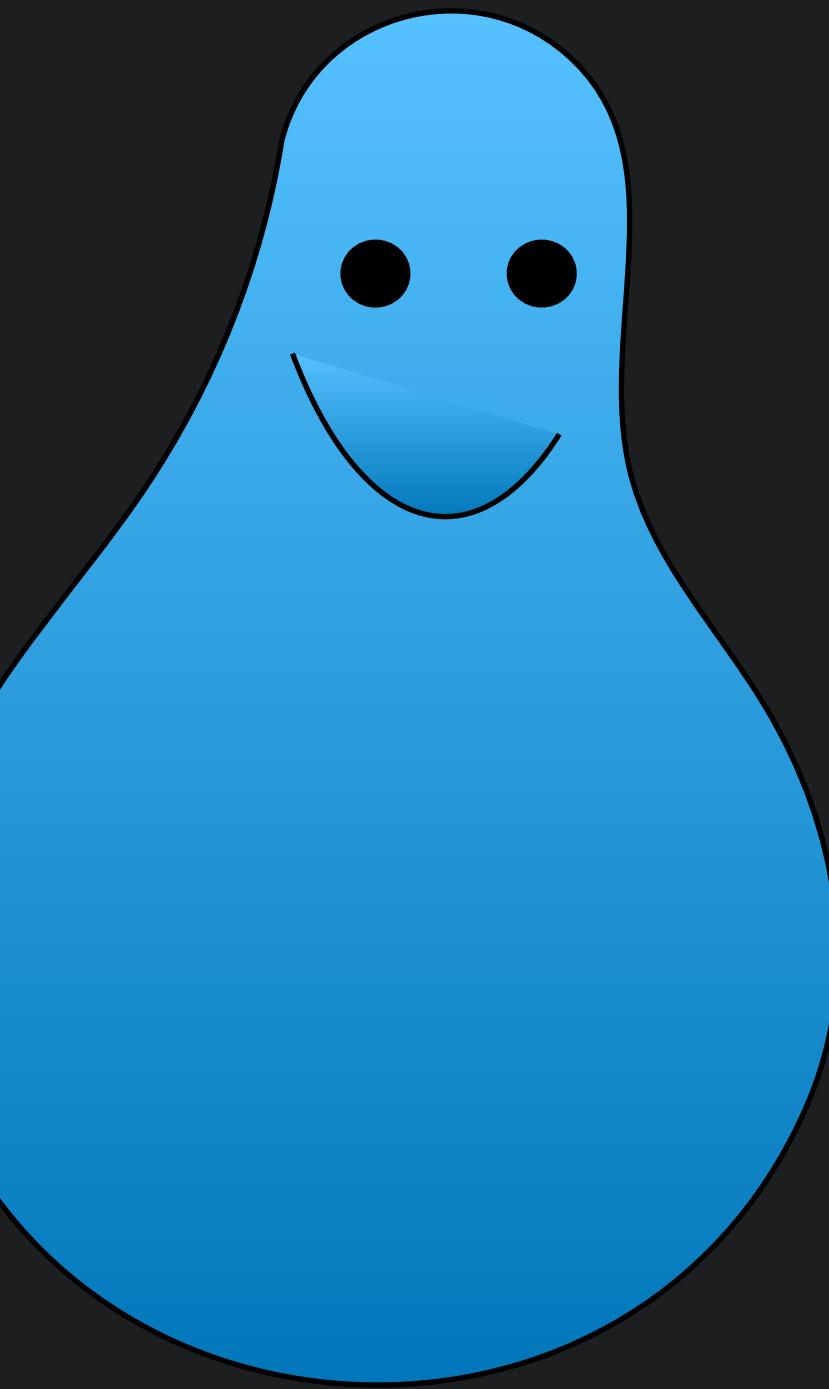
Blob

```
class Blob:  
    """  
    A Blob is a living creature populating the Forest.  
    Each blob can be either alive or dead, depending on its ability to search for food.  
    If a Blob eats enough food during a day, it survives to the next day. Otherwise, it dies.  
    """
```

The Blob only search for food during Daylight. When the nighttime arrives, it goes to sleep.

```
"""  
def __init__(self, env, forest, name):  
    self.env = env  
    self.name = name # Name of the blob  
  
    self.alive = True # Denotes the Blob's state: alive or dead  
    self.isHunting = True # Denotes if the Blob is searching for food  
    self.food_requirements = 1 # Minimum food to eat per day to survive  
    self.food_eaten = 0 # Amount of food eaten during the current day  
    self.can_reproduce = False  
  
    self.sleep_proc = env.process(self.sleep(env, forest))  
    self.hunt_proc = env.process(self.hunt(env, forest))
```

Various attributes serving as flags in if and for statements



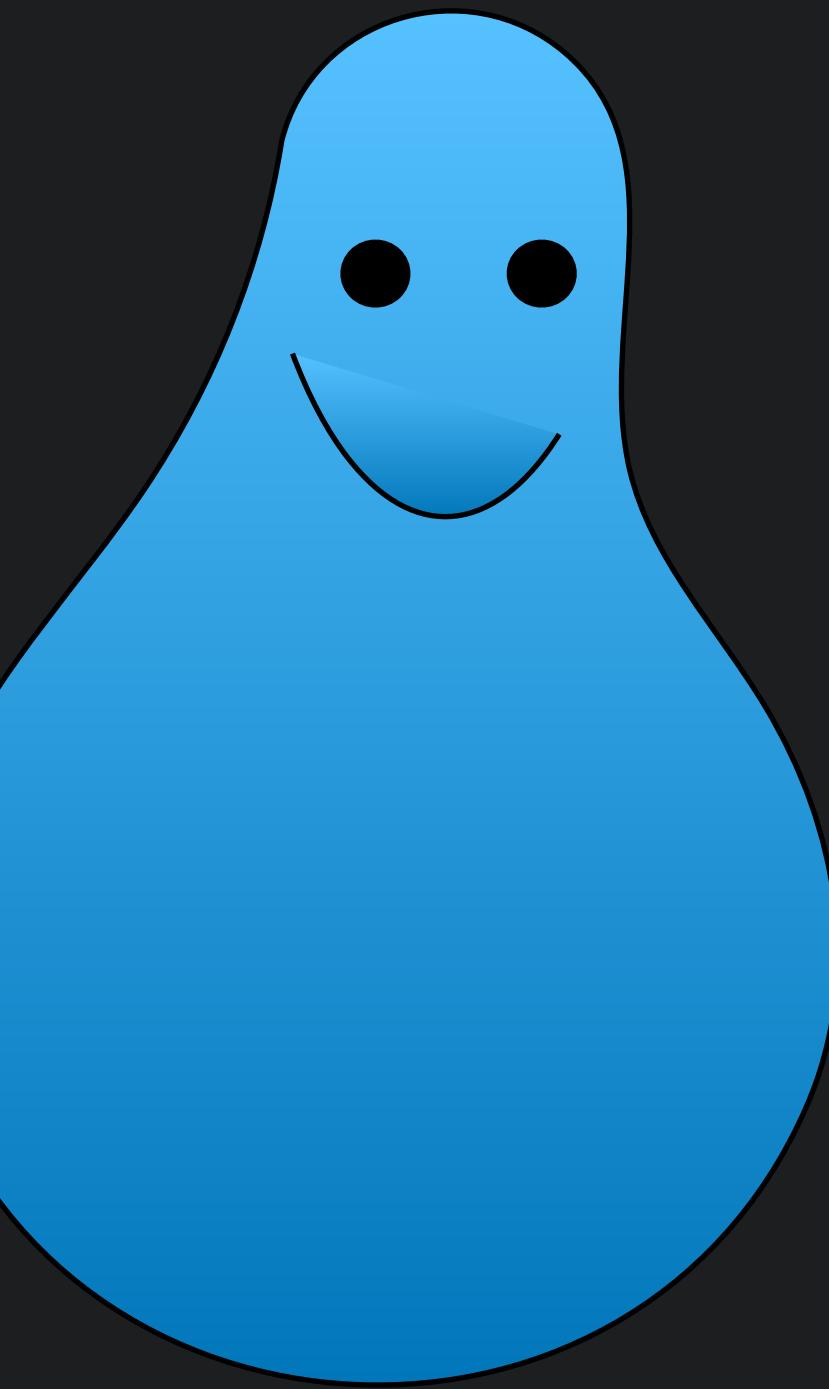
Blob

```
class Blob:  
    """  
        A Blob is a living creature populating the Forest.  
        Each blob can be either alive or dead, depending on its ability to search for food.  
        If a Blob eats enough food during a day, it survives to the next day. Otherwise, it dies.  
    """
```

The Blob only search for food during Daylight. When the nighttime arrives, it goes to sleep.

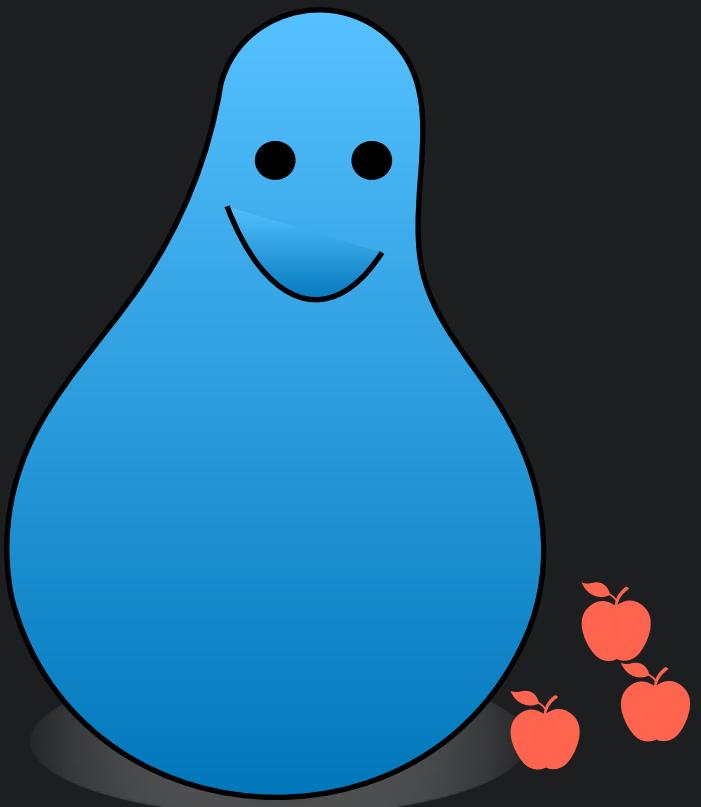
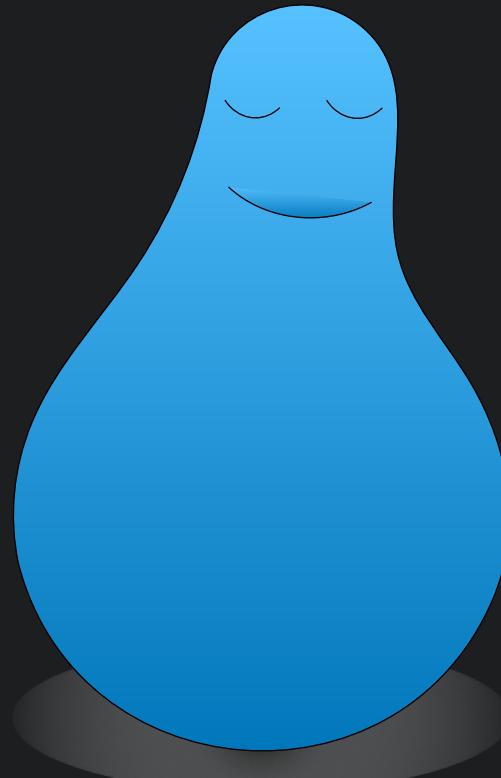
```
"""  
def __init__(self, env, forest, name):  
    self.env = env  
    self.name = name # Name of the blob  
  
    self.alive = True # Denotes the Blob's state: alive or dead  
    self.isHunting = True # Denotes if the Blob is searching for food  
    self.food_requirements = 1 # Minimum food to eat per day to survive  
    self.food_eaten = 0 # Amount of food eaten during the current day  
    self.can_reproduce = False  
  
    self.sleep_proc = env.process(self.sleep(env, forest))  
    self.hunt_proc = env.process(self.hunt(env, forest))
```

Various attributes serving as flags in if and for statements



Start the hunt and sleep process

Blob



```
def sleep(self, env, forest):
    """
    This method defines when the Blob goes to sleep, that is during Nighttime.
    It waits for the Nighttime to come, and then set the Blob to sleep.
    If the Blob is hunting while Nighttime starts, the hunting process is interrupted and the Blob
    goes to sleep.

    """
    while self.alive: # Execute only if the Blob is still alive
        yield env.timeout(DAYLIGHT) # Wait for the night to pass
        if self.isHunting: # If the Blob is hunting, interrupt it and set to NOT hunting.
            self.hunt_proc.interrupt()
            self.isHunting = False
            if self.food_eaten < self.food_requirements: # If the Blob hasn't eaten enough, it dies.
                self.alive = False
            if self.food_eaten > 3: # If the Blob ate more than 3 foods, then it reproduces creating a new Blob
                self.can_reproduce = True
            self.food_eaten = 0 # Reset to zero the food eaten for the next day

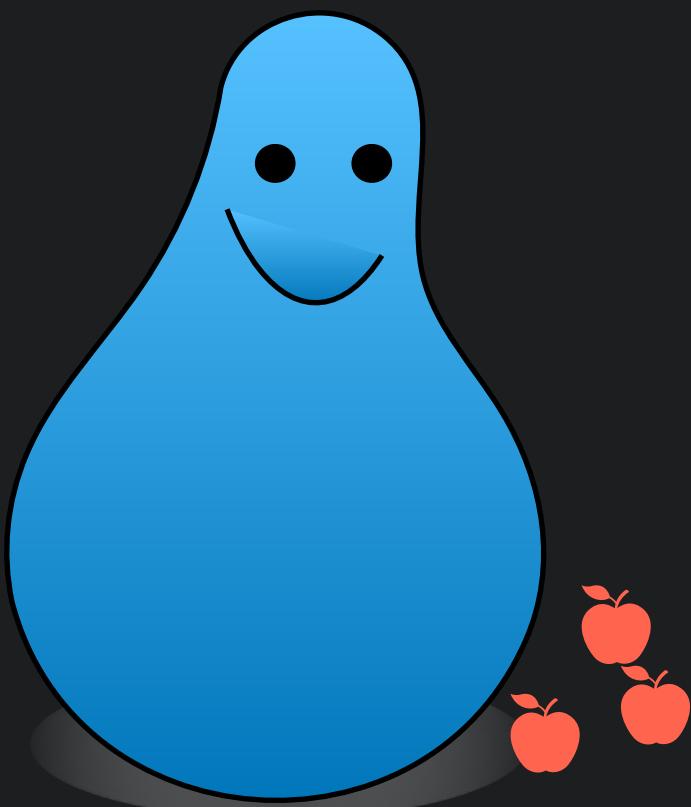
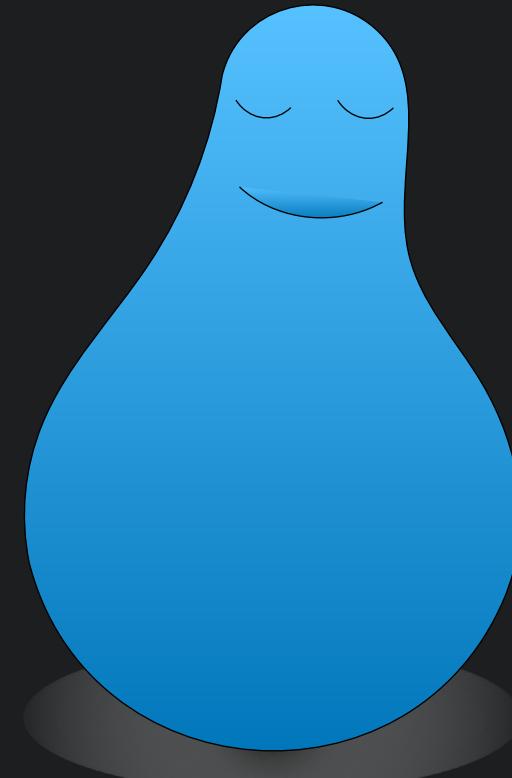
    def hunt(self, env, forest):
        """
        This method defines when the Blob goes searching fo food.

        """
        while self.alive: # Execute only if the Blob is still alive
            try:
                self.isHunting = True # Blob goes searching for food
                print(f'{self.name} starts hunting at time {env.now}')
                yield env.timeout(time_to_food()) # Some time is spent searching some food
                if forest.food.level > 0:
                    yield forest.food.get(1) # A food item is gathered from the Forest
                    self.food_eaten += 1 # The Blob has eaten a food
                else:
                    # print('Forest is depleted')
                    pass
            except simpy.Interrupt: # Nighttime has come
                self.isHunting = False
                yield env.timeout(NIGHTTIME) # Wait for the night to pass
            if self.can_reproduce:
                print("Newborn blob")
                Blob_list.append(Blob(env, forest, f'Newborn Blob {int(env.now/DAY)}'))
                self.can_reproduce = False
```

Blob

Prob. 0

why self.alive?



```
def sleep(self, env, forest):
    """
    This method defines when the Blob goes to sleep, that is during Nighttime.
    It waits for the Nighttime to come, and then set the Blob to sleep.
    If the Blob is hunting while Nighttime starts, the hunting process is interrupted and the Blob
    goes to sleep.

    """
    while self.alive: # Execute only if the Blob is still alive
        yield env.timeout(DAYLIGHT) # Wait for the night to pass
        if self.isHunting: # If the Blob is hunting, interrupt it and set to NOT hunting.
            self.hunt_proc.interrupt()
            self.isHunting = False
            if self.food_eaten < self.food_requirements: # If the Blob hasn't eaten enough, it dies.
                self.alive = False
            if self.food_eaten > 3: # If the Blob ate more than 3 foods, then it reproduces creating a new Blob
                self.can_reproduce = True
            self.food_eaten = 0 # Reset to zero the food eaten for the next day
```

```
def hunt(self, env, forest):
    """
    This method defines when the Blob goes searching fo food.

    """
    while self.alive: # Execute only if the Blob is still alive
```

```
        try:
            self.isHunting = True # Blob goes searching for food
            print(f'{self.name} starts hunting at time {env.now}')
            yield env.timeout(time_to_food()) # Some time is spent searching some food
            if forest.food.level > 0:
                yield forest.food.get(1) # A food item is gathered from the Forest
                self.food_eaten += 1 # The Blob has eaten a food
            else:
                # print('Forest is depleted')
                pass
        except simpy.Interrupt: # Nighttime has come
            self.isHunting = False
            yield env.timeout(NIGHTTIME) # Wait for the night to pass
            if self.can_reproduce:
                print("Newborn blob")
                Blob_list.append(Blob(env, forest, f'Newborn Blob {int(env.now/DAY)}'))
                self.can_reproduce = False
```

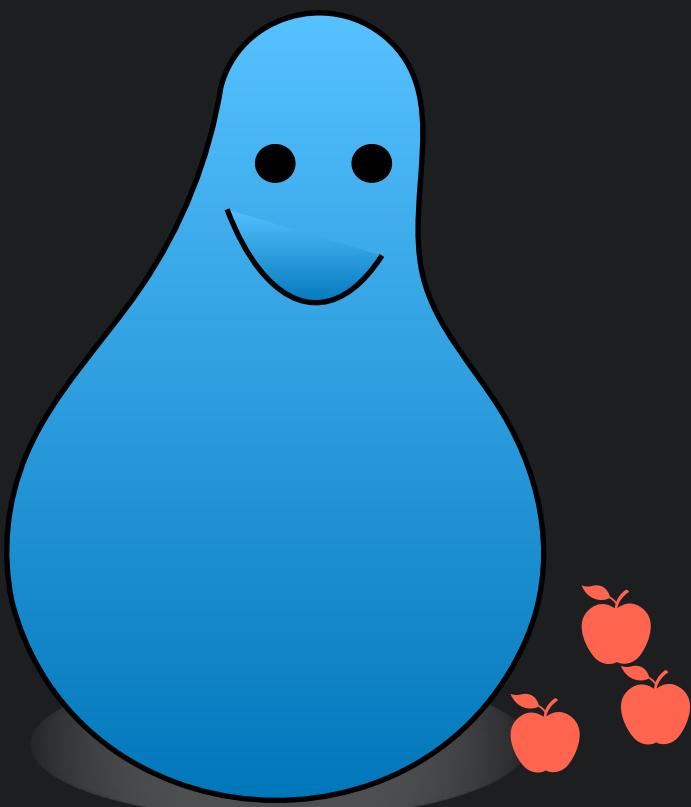
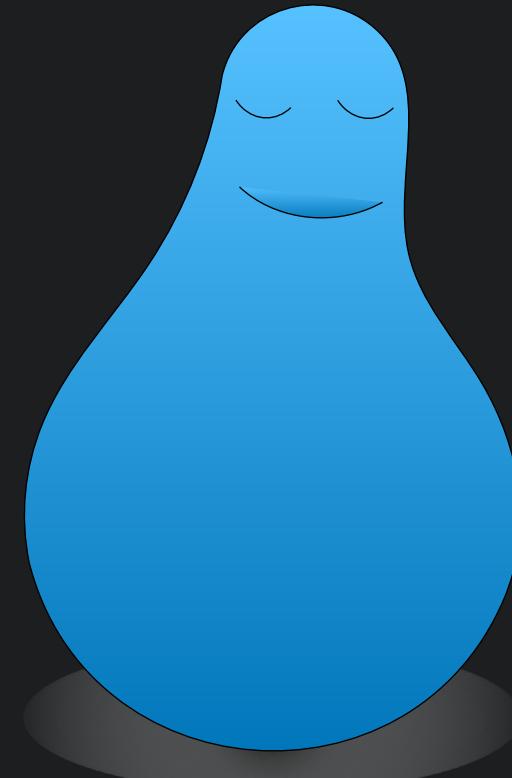
Blob

Prob. 0

why self.alive?

Prob. 1

why self.isHunting?



```
def sleep(self, env, forest):
    """
    This method defines when the Blob goes to sleep, that is during Nighttime.
    It waits for the Nighttime to come, and then set the Blob to sleep.
    If the Blob is hunting while Nighttime starts, the hunting process is interrupted and the Blob
    goes to sleep.

    """
    while self.alive: # Execute only if the Blob is still alive
        yield env.timeout(DAYLIGHT) # Wait for the night to pass
        if self.isHunting: # If the Blob is hunting, interrupt it and set to NOT hunting.
            self.hunt_proc.interrupt()
            self.isHunting = False
        if self.food_eaten < self.food_requirements: # If the Blob hasn't eaten enough, it dies.
            self.alive = False
        if self.food_eaten > 3: # If the Blob ate more than 3 foods, then it reproduces creating a new Blob
            self.can_reproduce = True
        self.food_eaten = 0 # Reset to zero the food eaten for the next day
```

```
def hunt(self, env, forest):
    """
    This method defines when the Blob goes searching fo food.
    """


```

```
    while self.alive: # Execute only if the Blob is still alive
        try:
            self.isHunting = True # Blob goes searching for food
            print(f'{self.name} starts hunting at time {env.now}')
            yield env.timeout(time_to_food()) # Some time is spent searching some food
            if forest.food.level > 0:
                yield forest.food.get(1) # A food item is gathered from the Forest
                self.food_eaten += 1 # The Blob has eaten a food
            else:
                # print('Forest is depleted')
                pass
        except simpy.Interrupt: # Nighttime has come
            self.isHunting = False
            yield env.timeout(NIGHTTIME) # Wait for the night to pass
        if self.can_reproduce:
            print("Newborn blob")
            Blob_list.append(Blob(env, forest, f'Newborn Blob {int(env.now/DAY)}'))
            self.can_reproduce = False
```

Blob

Prob. 0

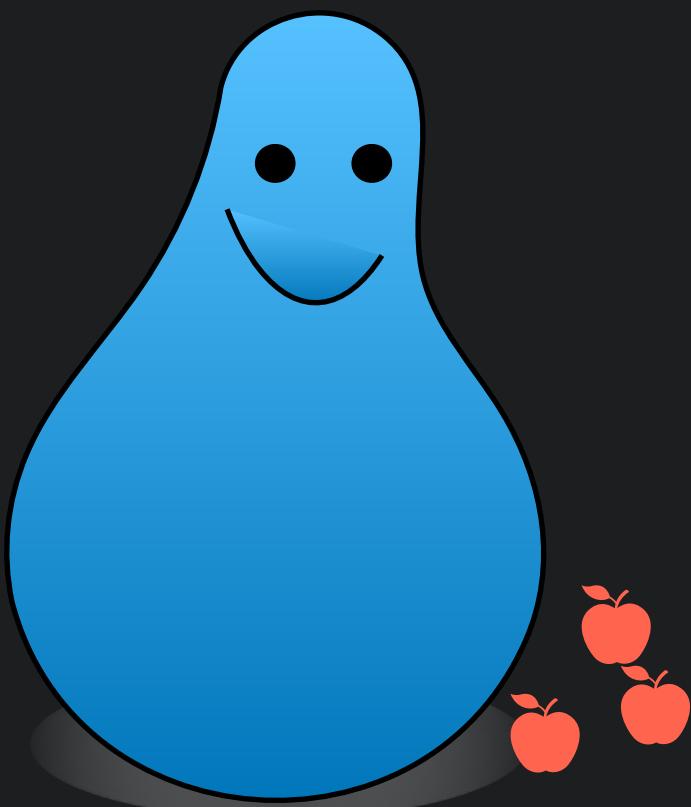
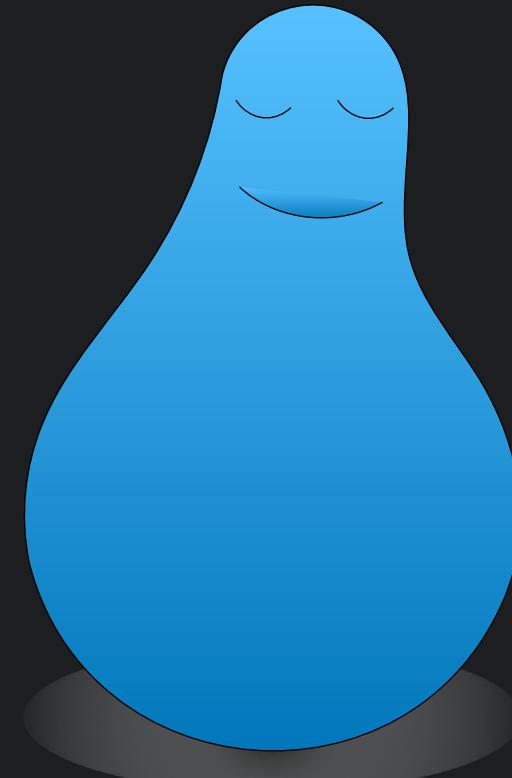
why self.alive?

Prob. 1

why self.isHunting?

Prob. 2

why self.can_reproduce?



```
def sleep(self, env, forest):
    """
    This method defines when the Blob goes to sleep, that is during Nighttime.
    It waits for the Nighttime to come, and then set the Blob to sleep.
    If the Blob is hunting while Nighttime starts, the hunting process is interrupted and the Blob
    goes to sleep.

    """
    while self.alive: # Execute only if the Blob is still alive
        yield env.timeout(DAYLIGHT) # Wait for the night to pass
        if self.isHunting: # If the Blob is hunting, interrupt it and set to NOT hunting.
            self.hunt_proc.interrupt()
            self.isHunting = False
        if self.food_eaten < self.food_requirements: # If the Blob hasn't eaten enough, it dies.
            self.alive = False
        if self.food_eaten > 3: # If the Blob ate more than 3 foods, then it reproduces creating a new Blob
            self.can_reproduce = True
        self.food_eaten = 0 # Reset to zero the food eaten for the next day
```

```
def hunt(self, env, forest):
    """
    This method defines when the Blob goes searching fo food.

    """
    while self.alive: # Execute only if the Blob is still alive
        try:
            self.isHunting = True # Blob goes searching for food
            print(f'{self.name} starts hunting at time {env.now}')
            yield env.timeout(time_to_food()) # Some time is spent searching some food
            if forest.food.level > 0:
                yield forest.food.get(1) # A food item is gathered from the Forest
                self.food_eaten += 1 # The Blob has eaten a food
            else:
                # print('Forest is depleted')
                pass
        except simpy.Interrupt: # Nighttime has come
            self.isHunting = False
            yield env.timeout(NIGHTTIME) # Wait for the night to pass
        if self.can_reproduce:
            print("Newborn blob")
            Blob_list.append(Blob(env, forest, f'Newborn Blob {int(env.now/DAY)}'))
            self.can_reproduce = False
```

Blob

Prob. 0

why self.alive?

Prob. 1

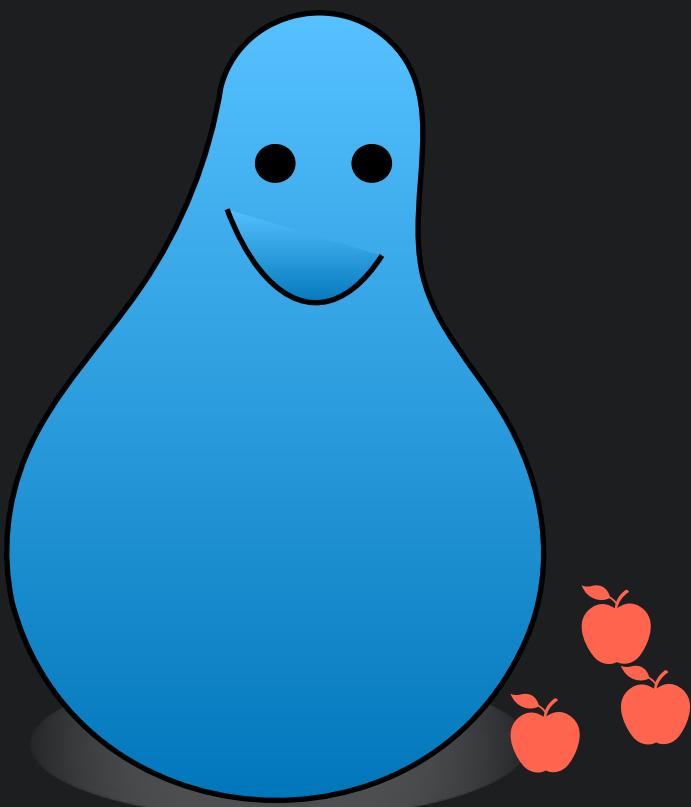
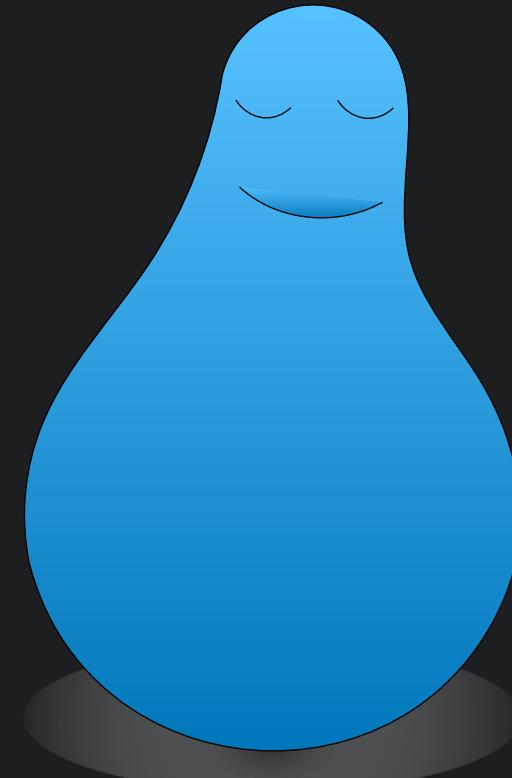
why self.isHunting?

Prob. 2

why self.can_reproduce?

Prob. 3

the .get() method



```
def sleep(self, env, forest):
    """
    This method defines when the Blob goes to sleep, that is during Nighttime.
    It waits for the Nighttime to come, and then set the Blob to sleep.
    If the Blob is hunting while Nighttime starts, the hunting process is interrupted and the Blob
    goes to sleep.

    """
    while self.alive: # Execute only if the Blob is still alive
        yield env.timeout(DAYLIGHT) # Wait for the night to pass
        if self.isHunting: # If the Blob is hunting, interrupt it and set to NOT hunting.
            self.hunt_proc.interrupt()
            self.isHunting = False
        if self.food_eaten < self.food_requirements: # If the Blob hasn't eaten enough, it dies.
            self.alive = False
        if self.food_eaten > 3: # If the Blob ate more than 3 foods, then it reproduces creating a new Blob
            self.can_reproduce = True
        self.food_eaten = 0 # Reset to zero the food eaten for the next day
```

```
def hunt(self, env, forest):
    """
    This method defines when the Blob goes searching fo food.

    """
    while self.alive: # Execute only if the Blob is still alive
        try:
            self.isHunting = True # Blob goes searching for food
            print(f'{self.name} starts hunting at time {env.now}')
            yield env.timeout(time_to_food()) # Some time is spent searching some food
            if forest.food.level > 0:
                yield forest.food.get(1) # A food item is gathered from the Forest
                self.food_eaten += 1 # The Blob has eaten a food
            else:
                # print('Forest is depleted')
                pass
        except simpy.Interrupt: # Nighttime has come
            self.isHunting = False
            yield env.timeout(NIGHTTIME) # Wait for the night to pass
        if self.can_reproduce:
            print("Newborn blob")
            Blob_list.append(Blob(env, forest, f'Newborn Blob {int(env.now/DAY)}'))
            self.can_reproduce = False
```

Blob

Prob. 0

why self.alive?

Prob. 1

why self.isHunting?

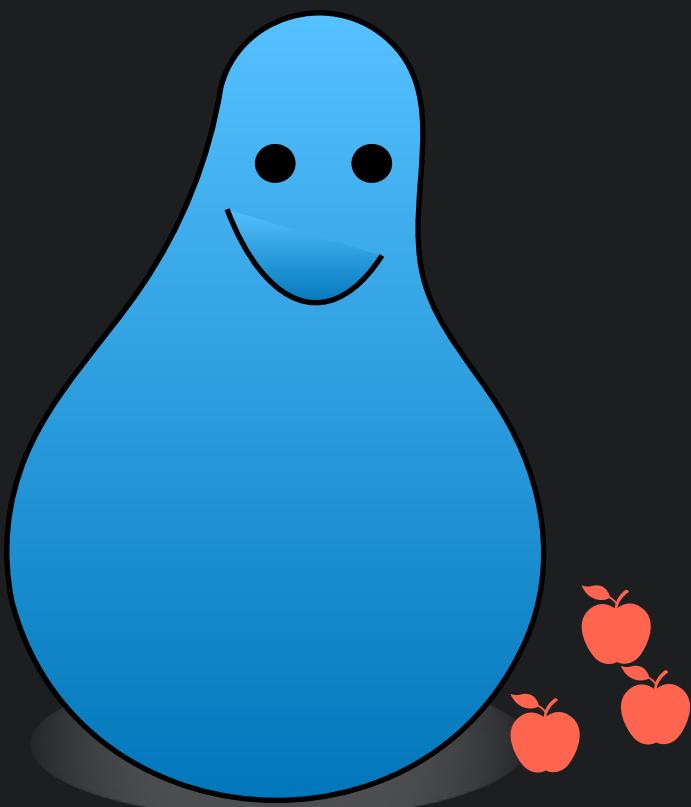
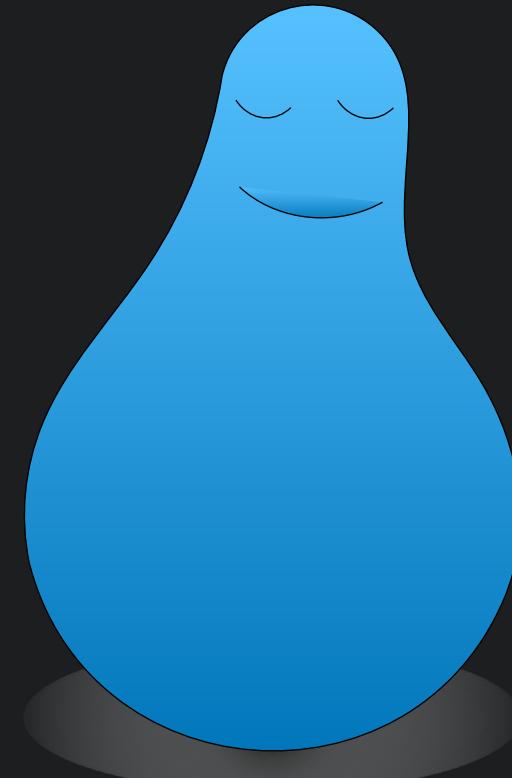
Prob. 2

why self.can_reproduce?

Prob. 3

the .get() method

Process interrupted
and exception



```
def sleep(self, env, forest):
    """
    This method defines when the Blob goes to sleep, that is during Nighttime.
    It waits for the Nighttime to come, and then set the Blob to sleep.
    If the Blob is hunting while Nighttime starts, the hunting process is interrupted and the Blob
    goes to sleep.

    """
    while self.alive: # Execute only if the Blob is still alive
        yield env.timeout(DAYLIGHT) # Wait for the night to pass
        if self.isHunting: # If the Blob is hunting, interrupt it and set to NOT hunting.
            self.hunt_proc.interrupt()
            self.isHunting = False
        if self.food_eaten < self.food_requirements: # If the Blob hasn't eaten enough, it dies.
            self.alive = False
        if self.food_eaten > 3: # If the Blob ate more than 3 foods, then it reproduces creating a new Blob
            self.can_reproduce = True
        self.food_eaten = 0 # Reset to zero the food eaten for the next day
```

```
def hunt(self, env, forest):
    """
    This method defines when the Blob goes searching fo food.

    """
    while self.alive: # Execute only if the Blob is still alive
        try:
            self.isHunting = True # Blob goes searching for food
            print(f'{self.name} starts hunting at time {env.now}')
            yield env.timeout(time_to_food()) # Some time is spent searching some food
            if forest.food.level > 0:
                yield forest.food.get(1) # A food item is gathered from the Forest
                self.food_eaten += 1 # The Blob has eaten a food
            else:
                # print('Forest is depleted')
                pass
        except simpy.Interrupt: # Nighttime has come
            self.isHunting = False
            yield env.timeout(NIGHTTIME) # Wait for the night to pass
        if self.can_reproduce:
            print("Newborn blob")
            Blob_list.append(Blob(env, forest, f'Newborn Blob {int(env.now/DAY)}'))
            self.can_reproduce = False
```

Blob

Prob. 0

why self.alive?

Prob. 1

why self.isHunting?

Prob. 2

why self.can_reproduce?

Prob. 3

the .get() method

Process interrupted
and exception

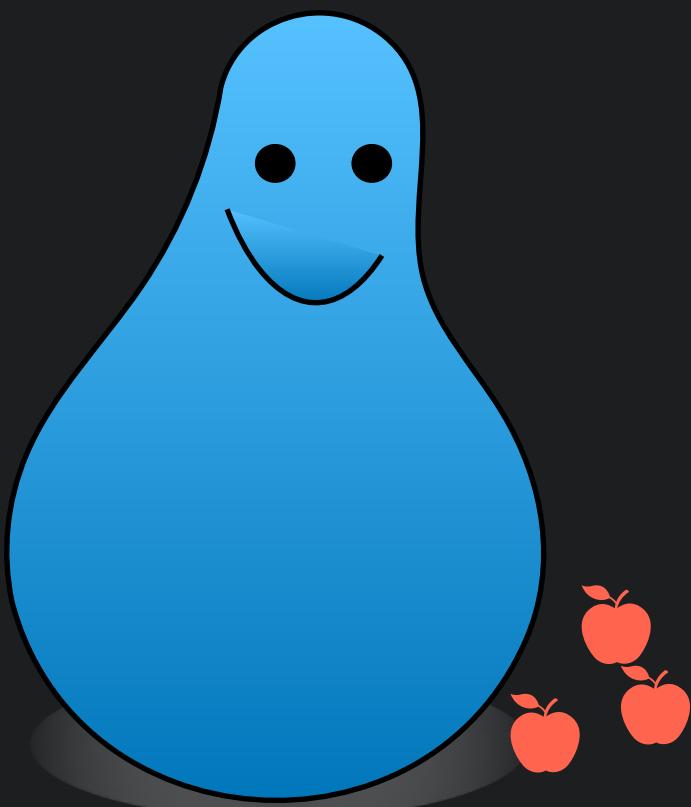
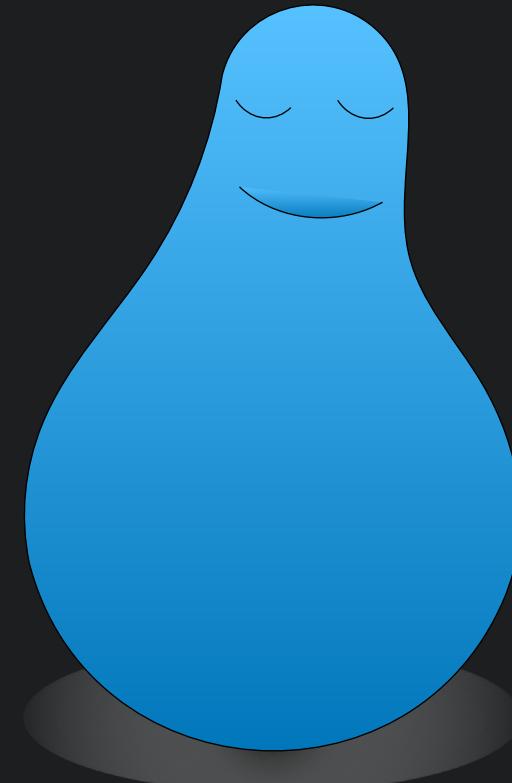
Newborn Blob!

```
def sleep(self, env, forest):
    """
    This method defines when the Blob goes to sleep, that is during Nighttime.
    It waits for the Nighttime to come, and then set the Blob to sleep.
    If the Blob is hunting while Nighttime starts, the hunting process is interrupted and the Blob
    goes to sleep.

    """
    while self.alive: # Execute only if the Blob is still alive
        yield env.timeout(DAYLIGHT) # Wait for the night to pass
        if self.isHunting: # If the Blob is hunting, interrupt it and set to NOT hunting.
            self.hunt_proc.interrupt()
            self.isHunting = False
        if self.food_eaten < self.food_requirements: # If the Blob hasn't eaten enough, it dies.
            self.alive = False
        if self.food_eaten > 3: # If the Blob ate more than 3 foods, then it reproduces creating a new Blob
            self.can_reproduce = True
        self.food_eaten = 0 # Reset to zero the food eaten for the next day
```

```
def hunt(self, env, forest):
    """
    This method defines when the Blob goes searching fo food.

    """
    while self.alive: # Execute only if the Blob is still alive
        try:
            self.isHunting = True # Blob goes searching for food
            print(f'{self.name} starts hunting at time {env.now}')
            yield env.timeout(time_to_food()) # Some time is spent searching some food
            if forest.food.level > 0:
                yield forest.food.get(1) # A food item is gathered from the Forest
                self.food_eaten += 1 # The Blob has eaten a food
            else:
                # print('Forest is depleted')
                pass
        except simpy.Interrupt: # Nighttime has come
            self.isHunting = False
            yield env.timeout(NIGHTTIME) # Wait for the night to pass
            if self.can_reproduce:
                print("Newborn blob")
                Blob_list.append(Blob(env, forest, f'Newborn Blob {int(env.now/DAY)}'))
                self.can_reproduce = False
```



Main

```
def main(start_blob = 50, start_food = 1000, food_rate = 100, num_days = 50):

    global FOOD_RATE_PRODUCTION
    global NUM_BLOB
    global INITIAL_FOOD
    global NUM_DAYS

    NUM_BLOB = start_blob
    INITIAL_FOOD = start_food
    FOOD_RATE_PRODUCTION = food_rate
    NUM_DAYS = num_days

    # Declaration of the Simpy Environment, Forest and Blobs
    env = simply.Environment()
    forest = Forest(env)

    global Blob_list
    Blob_list = [Blob(env, forest, f'Blob {i}') for i in range(start_blob)]

    global living_blobs
    living_blobs = []

    global food_level_list
    food_level_list = []

    # Start the simulation
    env.run(until=NUM_DAYS*1440)
```

Main

Prob. 4 why global variables?

```
def main(start_blob = 50, start_food = 1000, food_rate = 100, num_days = 50):

    global FOOD_RATE_PRODUCTION
    global NUM_BLOB
    global INITIAL_FOOD
    global NUM_DAYS

    NUM_BLOB = start_blob
    INITIAL_FOOD = start_food
    FOOD_RATE_PRODUCTION = food_rate
    NUM_DAYS = num_days

    # Declaration of the Simpy Environment, Forest and Blobs
    env = simply.Environment()
    forest = Forest(env)

    global Blob_list
    Blob_list = [Blob(env, forest, f'Blob {i}') for i in range(start_blob)]

    global living_blobs
    living_blobs = []

    global food_level_list
    food_level_list = []

    # Start the simulation
    env.run(until=NUM_DAYS*1440)
```

Main

Prob. 4 why global variables?

```
def main(start_blob = 50, start_food = 1000, food_rate = 100, num_days = 50):

    global FOOD_RATE_PRODUCTION
    global NUM_BLOB
    global INITIAL_FOOD
    global NUM_DAYS

    NUM_BLOB = start_blob
    INITIAL_FOOD = start_food
    FOOD_RATE_PRODUCTION = food_rate
    NUM_DAYS = num_days

    # Declaration of the Simpy Environment, Forest and Blobs
    env = simpy.Environment()
    forest = Forest(env)

    global Blob_list
    Blob_list = [Blob(env, forest, f'Blob {i}') for i in range(start_blob)]

    global living_blobs
    living_blobs = []

    global food_level_list
    food_level_list = []

    # Start the simulation
    env.run(until=NUM_DAYS*1440)
```

Declaration for Simpy environment

Main

Prob. 4 why global variables?

Initial population of the forest

```
def main(start_blob = 50, start_food = 1000, food_rate = 100, num_days = 50):

    global FOOD_RATE_PRODUCTION
    global NUM_BLOB
    global INITIAL_FOOD
    global NUM_DAYS

    NUM_BLOB = start_blob
    INITIAL_FOOD = start_food
    FOOD_RATE_PRODUCTION = food_rate
    NUM_DAYS = num_days

    # Declaration of the Simpy Environment, Forest and Blobs
    env = simpy.Environment()
    forest = Forest(env)

    global Blob_list
    Blob_list = [Blob(env, forest, f'Blob {i}') for i in range(start_blob)]

    global living_blobs
    living_blobs = []

    global food_level_list
    food_level_list = []

    # Start the simulation
    env.run(until=NUM_DAYS*1440)
```

Declaration for Simpy environment

Main

Prob. 4 why global variables?

Initial population of the forest

```
def main(start_blob = 50, start_food = 1000, food_rate = 100, num_days = 50):

    global FOOD_RATE_PRODUCTION
    global NUM_BLOB
    global INITIAL_FOOD
    global NUM_DAYS

    NUM_BLOB = start_blob
    INITIAL_FOOD = start_food
    FOOD_RATE_PRODUCTION = food_rate
    NUM_DAYS = num_days

    # Declaration of the Simpy Environment, Forest and Blobs
    env = simpy.Environment()
    forest = Forest(env)

    global Blob_list
    Blob_list = [Blob(env, forest, f'Blob {i}') for i in range(start_blob)]

    global living_blobs
    living_blobs = []

    global food_level_list
    food_level_list = []

    # Start the simulation
    env.run(until=NUM_DAYS*1440)
```

Declaration for Simpy environment

Empty arrays for results

Main

Prob. 4
why global variables?

Initial population of the forest

Simulation starts and lasts for NUM_DAYS

```
def main(start_blob = 50, start_food = 1000, food_rate = 100, num_days = 50):

    global FOOD_RATE_PRODUCTION
    global NUM_BLOB
    global INITIAL_FOOD
    global NUM_DAYS

    NUM_BLOB = start_blob
    INITIAL_FOOD = start_food
    FOOD_RATE_PRODUCTION = food_rate
    NUM_DAYS = num_days

    # Declaration of the Simpy Environment, Forest and Blobs
    env = simpy.Environment()
    forest = Forest(env)

    global Blob_list
    Blob_list = [Blob(env, forest, f'Blob {i}') for i in range(start_blob)]

    global living_blobs
    living_blobs = []

    global food_level_list
    food_level_list = []

    # Start the simulation
    env.run(until=NUM_DAYS*1440)
```

Declaration for Simpy environment

Empty arrays for results

Main

Prob. 4
why global variables?

Initial population of the forest

Simulation starts and lasts for NUM_DAYS

```
def main(start_blob = 50, start_food = 1000, food_rate = 100, num_days = 50):

    global FOOD_RATE_PRODUCTION
    global NUM_BLOB
    global INITIAL_FOOD
    global NUM_DAYS

    NUM_BLOB = start_blob
    INITIAL_FOOD = start_food
    FOOD_RATE_PRODUCTION = food_rate
    NUM_DAYS = num_days

    # Declaration of the Simpy Environment, Forest and Blobs
    env = simpy.Environment()
    forest = Forest(env)

    global Blob_list
    Blob_list = [Blob(env, forest, f'Blob {i}') for i in range(start_blob)]

    global living_blobs
    living_blobs = []

    global food_level_list
    food_level_list = []

    # Start the simulation
    env.run(until=NUM_DAYS*1440)
```

Declaration for Simpy environment

Empty arrays for results

(if __name__ == '__main__': main() Snippet for executing as script)

SIMULATION RESULTS

Initial Blob: 50
Blob survived: 56
Initial food: 1000
Food growth: 100
Number of days: 50

Living Blobs [50, 52, 56, 58, 60, 62, 67, 71, 73, 74, 79, 80, 83, 87, 90, 93, 94, 100, 101, 107, 113, 119, 125, 130, 82, 70, 64, 66, 63, 62, 65, 62, 60, 56, 55, 57, 60, 63, 66, 68, 64, 65, 68, 64, 61, 62, 64, 62, 63, 61]

Food Level [1000, 1019, 1031, 1034, 1041, 1032, 1032, 1010, 991, 974, 951, 923, 890, 852, 807, 764, 711, 633, 579, 497, 414, 321, 210, 100, 100, 10, 100, 101, 100, 100, 103, 100, 100, 100, 112, 119, 115, 105, 103, 100, 104, 100, 100, 100, 102, 101, 100, 100, 100]

SIMULATION RESULTS

Initial Blob: 50
Blob survived: 56
Initial food: 1000
Food growth: 100
Number of days: 50

Living Blobs [50, 52, 56, 58, 60, 62, 67, 71, 73, 74, 79, 80, 83, 87, 90, 93, 94, 100, 101, 107, 113, 119, 125, 130, 82, 70, 64, 66, 63, 62, 65, 62, 60, 56, 55, 57, 60, 63, 66, 68, 64, 65, 68, 64, 61, 62, 64, 62, 63, 61]

Food Level [1000, 1019, 1031, 1034, 1041, 1032, 1032, 1010, 991, 974, 951, 923, 890, 852, 807, 764, 711, 633, 579, 497, 414, 321, 210, 100, 100, 100, 101, 100, 100, 103, 100, 100, 100, 112, 119, 115, 105, 103, 100, 104, 100, 100, 100, 102, 101, 100, 100, 100]

Run 0

SIMULATION RESULTS

Initial Blob: 50
Blob survived: 62
Initial food: 1000
Food growth: 100
Number of days: 50

Living Blobs [50, 52, 53, 57, 60, 61, 63, 67, 69, 76, 81, 86, 86, 89, 93, 95, 95, 98, 100, 104, 109, 114, 118, 118, 73, 68, 63, 63, 63, 64, 66, 68, 66, 62, 63, 62, 61, 63, 58, 60, 62, 62, 62, 63, 65, 63, 64, 63, 61, 66]

Food Level [1000, 1021, 1042, 1046, 1049, 1056, 1045, 1035, 1025, 997, 959, 919, 886, 844, 791, 734, 675, 606, 551, 476, 390, 299, 209, 100, 100, 100, 100, 108, 112, 107, 105, 100, 100, 100, 100, 100, 111, 109, 100, 100, 102, 100, 100, 100, 100, 100]

Run 1

SIMULATION RESULTS

Initial Blob: 50
Blob survived: 56
Initial food: 1000
Food growth: 100
Number of days: 50

Living Blobs [50, 52, 56, 58, 60, 62, 67, 71, 73, 74, 79, 80, 83, 87, 90, 93, 94, 100, 101, 107, 113, 119, 125, 130, 82, 70, 64, 66, 63, 62, 65, 62, 60, 56, 55, 57, 60, 63, 66, 68, 64, 65, 68, 64, 61, 62, 64, 62, 63, 61]

Food Level [1000, 1019, 1031, 1034, 1041, 1032, 1032, 1010, 991, 974, 951, 923, 890, 852, 807, 764, 711, 633, 579, 497, 414, 321, 210, 100, 100, 100, 101, 100, 100, 103, 100, 100, 100, 112, 119, 115, 105, 103, 100, 104, 100, 100, 100, 102, 101, 100, 100, 100]

Run 0

SIMULATION RESULTS

Initial Blob: 50
Blob survived: 62
Initial food: 1000
Food growth: 100
Number of days: 50

Living Blobs [50, 52, 53, 57, 60, 61, 63, 67, 69, 76, 81, 86, 86, 89, 93, 95, 95, 98, 100, 104, 109, 114, 118, 118, 73, 68, 63, 63, 63, 64, 66, 68, 66, 62, 63, 62, 61, 63, 58, 60, 62, 62, 62, 63, 65, 63, 64, 63, 61, 66]

Food Level [1000, 1021, 1042, 1046, 1049, 1056, 1045, 1035, 1025, 997, 959, 919, 886, 844, 791, 734, 675, 606, 551, 476, 390, 299, 209, 100, 100, 100, 100, 108, 112, 107, 105, 100, 100, 100, 100, 100, 100, 100, 111, 109, 100, 100, 102, 100, 100, 100, 100, 100, 100]

Run 1

SIMULATION RESULTS

Initial Blob: 50
Blob survived: 60
Initial food: 1000
Food growth: 100
Number of days: 50

Living Blobs [50, 50, 55, 57, 59, 63, 67, 70, 73, 81, 84, 85, 86, 89, 95, 101, 105, 106, 111, 113, 114, 118, 119, 75, 68, 64, 64, 63, 64, 61, 60, 60, 61, 63, 62, 63, 62, 62, 58, 58, 59, 61, 61, 60, 63, 66, 67, 66, 65, 60]

Food Level [1000, 1014, 1020, 1027, 1024, 1015, 1000, 984, 970, 921, 883, 857, 819, 773, 723, 664, 601, 539, 450, 370, 278, 194, 100]

Run 2

averages.py

```
results = []
foods = []
# In this way the print statements in bs.main() are suppressed
with HiddenPrints():
    for i in tqdm(range(NUM_SIMULATIONS)):
        # results.append(np.array(bs.main(start_blob = NUM_BLOB, start_food = INITIAL_FOOD, food_rate = FOOD_RATE_PRODUCTION)))
        _res = bs.main(start_blob = NUM_BLOB, start_food = INITIAL_FOOD, food_rate = FOOD_RATE_PRODUCTION, num_days = NUM_DAYS)
        results.append(np.array(_res[0]))
        foods.append(np.array(_res[1]))

results = np.array(results)
foods = np.array(foods)

# Some processing of the data with Numpy builtin functions 'mean' ans 'std'
mean_blobs = np.mean(results, axis=0)
std_blobs = np.sqrt(np.var(results, axis = 0))

mean_foods = np.mean(foods, axis=0)
std_foods = np.sqrt(np.var(foods, axis=0))
```

```
class HiddenPrints:
    """
    Useful Context manager to suppress the print statements in called functions.
    To be used with the 'with' syntax (see code below).
    """
    def __enter__(self):
        self._original_stdout = sys.stdout
        sys.stdout = open(os.devnull, 'w')

    def __exit__(self, exc_type, exc_val, exc_tb):
        sys.stdout.close()
        sys.stdout = self._original_stdout
```

averages.py

Suppress printing

```
results = []
foods = []
# In this way the print statements in bs.main() are suppressed
with HiddenPrints():
    for i in tqdm(range(NUM_SIMULATIONS)):
        # results.append(np.array(bs.main(start_blob = NUM_BLOB, start_food = INITIAL_FOOD, food_rate = FOOD_RATE_PRODUCTION)))
        _res = bs.main(start_blob = NUM_BLOB, start_food = INITIAL_FOOD, food_rate = FOOD_RATE_PRODUCTION, num_days = NUM_DAYS)
        results.append(np.array(_res[0]))
        foods.append(np.array(_res[1]))

results = np.array(results)
foods = np.array(foods)

# Some processing of the data with Numpy builtin functions 'mean' ans 'std'
mean_blobs = np.mean(results, axis=0)
std_blobs = np.sqrt(np.var(results, axis = 0))

mean_foods = np.mean(foods, axis=0)
std_foods = np.sqrt(np.var(foods, axis=0))
```

```
class HiddenPrints:
    """
    Useful Context manager to suppress the print statements in called functions.
    To be used with the 'with' syntax (see code below).
    """
    def __enter__(self):
        self._original_stdout = sys.stdout
        sys.stdout = open(os.devnull, 'w')

    def __exit__(self, exc_type, exc_val, exc_tb):
        sys.stdout.close()
        sys.stdout = self._original_stdout
```

averages.py

```
results = []
foods = []
# In this way the print statements in bs.main() are suppressed
with HiddenPrints():
    for i in tqdm(range(NUM_SIMULATIONS)):
        # results.append(np.array(bs.main(start_blob = NUM_BLOB, start_food = INITIAL_FOOD, food_rate = FOOD_RATE_PRODUCTION)))
        _res = bs.main(start_blob = NUM_BLOB, start_food = INITIAL_FOOD, food_rate = FOOD_RATE_PRODUCTION, num_days = NUM_DAYS)
        results.append(np.array(_res[0]))
        foods.append(np.array(_res[1]))

results = np.array(results)
foods = np.array(foods)

# Some processing of the data with Numpy builtin functions 'mean' ans 'std'
mean_blobs = np.mean(results, axis=0)
std_blobs = np.sqrt(np.var(results, axis = 0))

mean_foods = np.mean(foods, axis=0)
std_foods = np.sqrt(np.var(foods, axis=0))
```

Suppress printing

```
class HiddenPrints:
    """
    Useful Context manager to suppress the print statements in called functions.
    To be used with the 'with' syntax (see code below).
    """
    def __enter__(self):
        self._original_stdout = sys.stdout
        sys.stdout = open(os.devnull, 'w')

    def __exit__(self, exc_type, exc_val, exc_tb):
        sys.stdout.close()
        sys.stdout = self._original_stdout
```

tqdm is a nice progress bar for Python

averages.py

```
results = []
foods = []
# In this way the print statements in bs.main() are suppressed
with HiddenPrints():
    for i in tqdm(range(NUM_SIMULATIONS)):
        # results.append(np.array(bs.main(start_blob = NUM_BLOB, start_food = INITIAL_FOOD, food_rate = FOOD_RATE_PRODUCTION)))
        _res = bs.main(start_blob = NUM_BLOB, start_food = INITIAL_FOOD, food_rate = FOOD_RATE_PRODUCTION, num_days = NUM_DAYS)
        results.append(np.array(_res[0]))
        foods.append(np.array(_res[1]))
results = np.array(results)
foods = np.array(foods)

# Some processing of the data with Numpy builtin functions 'mean' ans 'std'
mean_blobs = np.mean(results, axis=0)
std_blobs = np.sqrt(np.var(results, axis = 0))

mean_foods = np.mean(foods, axis=0)
std_foods = np.sqrt(np.var(foods, axis=0))
```

Suppress printing

```
class HiddenPrints:
    """
    Useful Context manager to suppress the print statements in called functions.
    To be used with the 'with' syntax (see code below).
    """
    def __enter__(self):
        self._original_stdout = sys.stdout
        sys.stdout = open(os.devnull, 'w')

    def __exit__(self, exc_type, exc_val, exc_tb):
        sys.stdout.close()
        sys.stdout = self._original_stdout
```

tqdm is a nice progress bar for Python

**Imports bio_sim.py and executes the simulation
NUM_SIMULATIONS times**

averages.py

```
results = []
foods = []
# In this way the print statements in bs.main() are suppressed
with HiddenPrints():
    for i in tqdm(range(NUM_SIMULATIONS)):
        # results.append(np.array(bs.main(start_blob = NUM_BLOB, start_food = INITIAL_FOOD, food_rate = FOOD_RATE_PRODUCTION)))
        _res = bs.main(start_blob = NUM_BLOB, start_food = INITIAL_FOOD, food_rate = FOOD_RATE_PRODUCTION, num_days = NUM_DAYS)
        results.append(np.array(_res[0]))
        foods.append(np.array(_res[1]))
results = np.array(results)
foods = np.array(foods)

# Some processing of the data with Numpy builtin functions 'mean' ans 'std'
mean_blobs = np.mean(results, axis=0)
std_blobs = np.sqrt(np.var(results, axis = 0))

mean_foods = np.mean(foods, axis=0)
std_foods = np.sqrt(np.var(foods, axis=0))
```

Suppress printing

```
class HiddenPrints:
    """
    Useful Context manager to suppress the print statements in called functions.
    To be used with the 'with' syntax (see code below).
    """

    def __enter__(self):
        self._original_stdout = sys.stdout
        sys.stdout = open(os.devnull, 'w')

    def __exit__(self, exc_type, exc_val, exc_tb):
        sys.stdout.close()
        sys.stdout = self._original_stdout
```

tqdm is a nice progress bar for Python

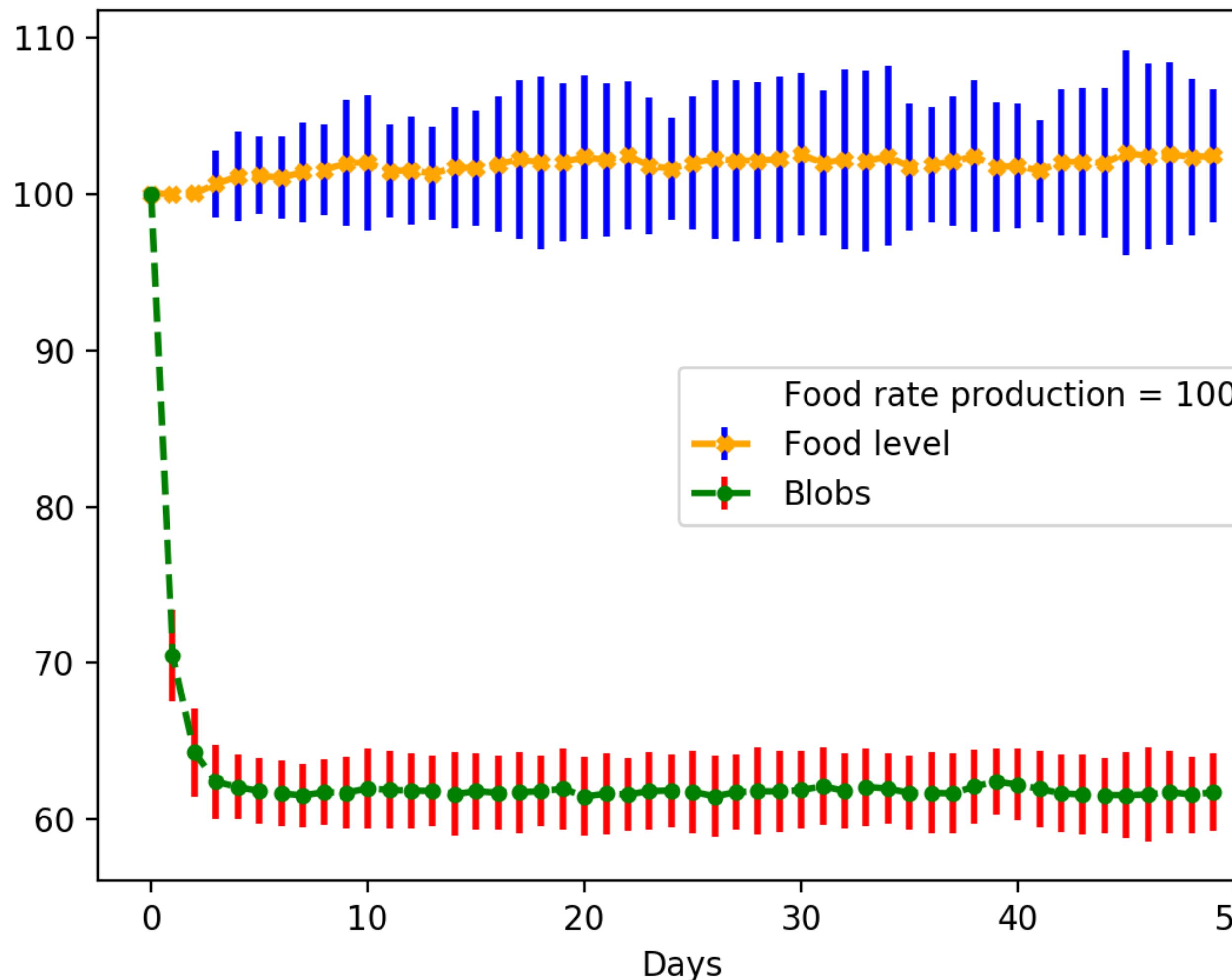
**Imports bio_sim.py and executes the simulation
NUM_SIMULATIONS times**

**Basic analysis of the results using Numpy:
mean and standard deviation**

averages.py

```
# PLOT OF RESULTS
if plotting:
    # plt.hlines(FOOD_RATE_PRODUCTION, xmin=0, xmax=30, linestyle='dashed', linewidth=1, label='Food rate production')
    # plt.hlines(INITIAL_FOOD, xmin=0, xmax=30, linestyle='dashed', linewidth=1, label='Initial Food Level')
    plt.plot([],[], linewidth = 0, label=f'Food rate production = {FOOD_RATE_PRODUCTION}')
    plt.errorbar(np.arange(NUM_DAYS), mean_foods, yerr=std_foods, color = 'orange', marker = 'X', linestyle='dashed', linewidth=2, markersize=4)
    plt.errorbar(np.arange(NUM_DAYS), mean_blobs, yerr=std_blobs, color='green', marker='o', linestyle='dashed', linewidth=2, markersize=4)
    plt.legend()
    plt.show()
```

averages.py

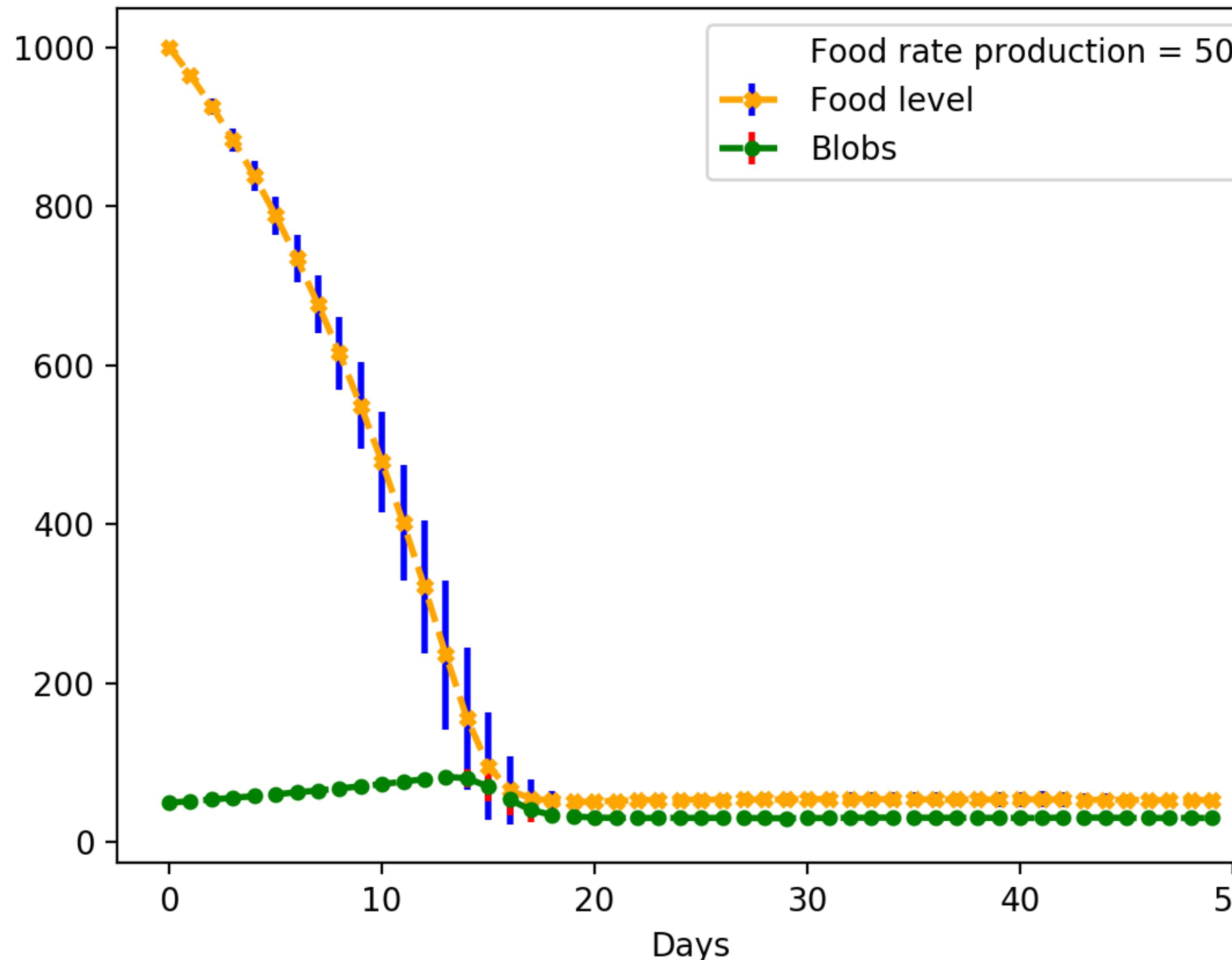


Initial population: 100

Initial Food: 100

Food rate production: 100

averages.py

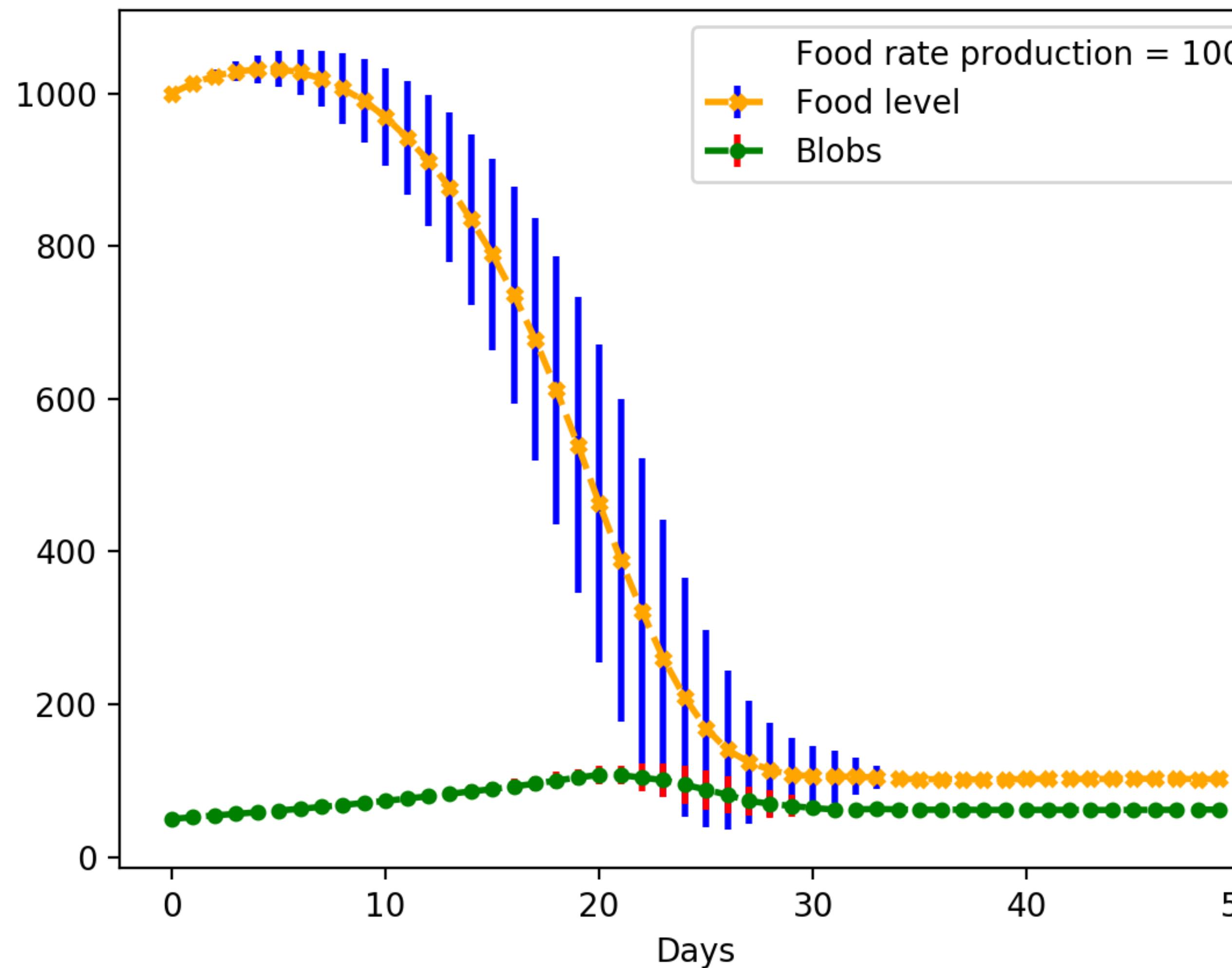


Initial population: 50

Initial Food: 1000

Food rate production: 50

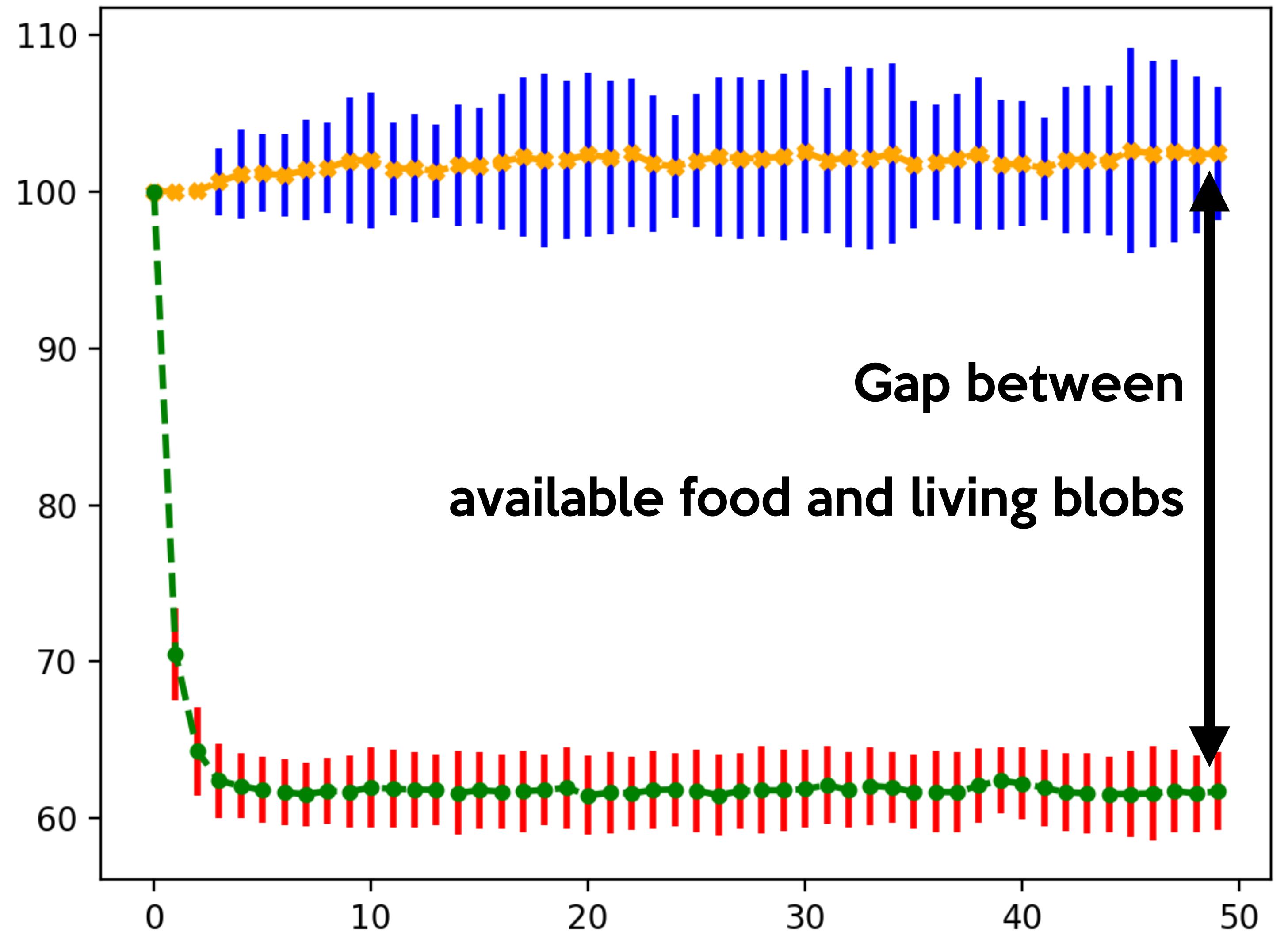
averages.py



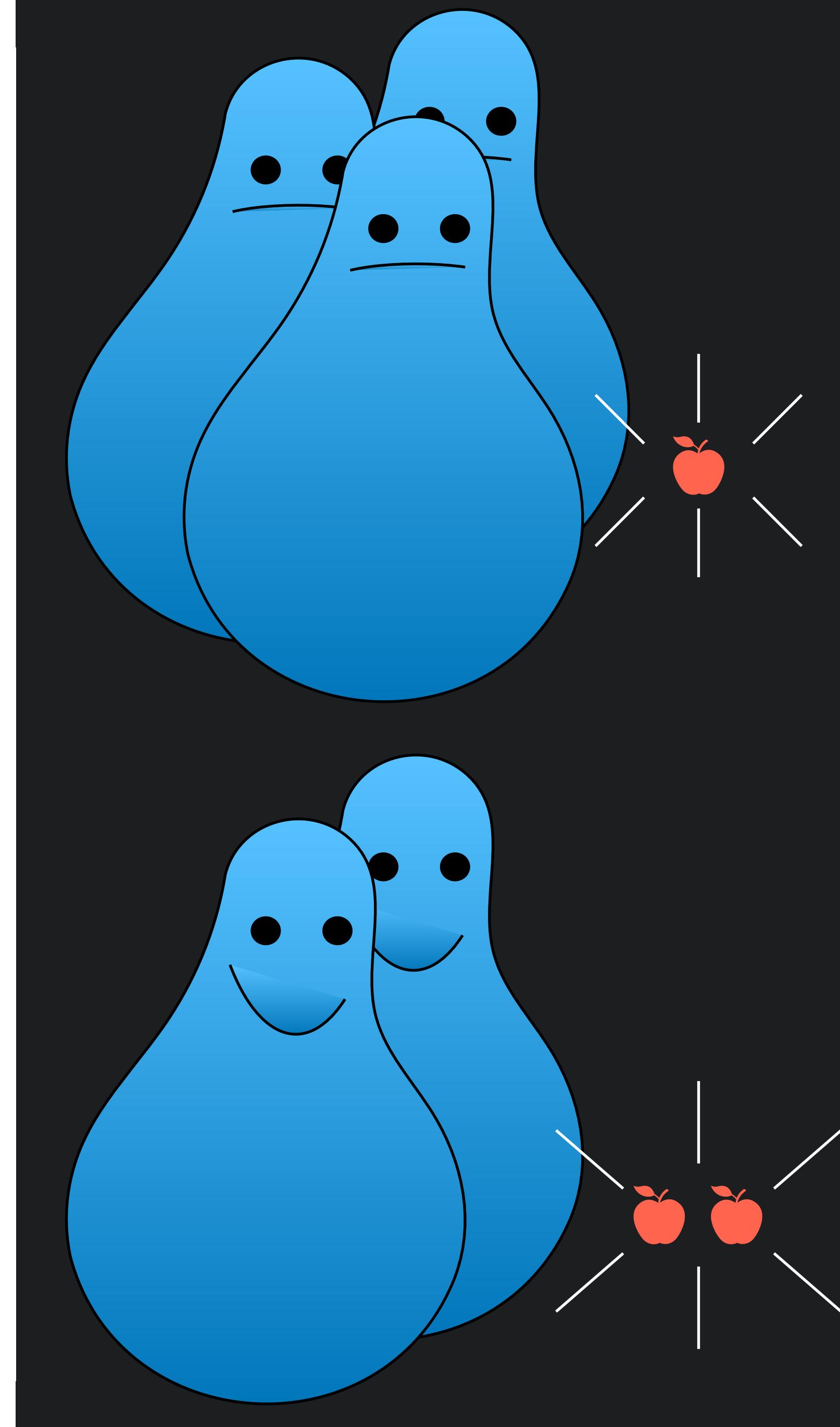
Initial population: 50

Initial Food: 1000

Food rate production: 100



Gap between
available food and living blobs



equilibrium.py

```
### Run the main.py file multiple times changing the food_rate_production parameters, and saves the ending food level and living blob Number
### for further plotting and analysis

results = []
foods = []
food_rates_list = [10*i for i in range(1,20)]
# In this way the print statements in bs.main() are suppressed
with HiddenPrints():
    for foodrate in tqdm(food_rates_list):
        # results.append(np.array(bs.main(start_blob = NUM_BLOB, start_food = INITIAL_FOOD, food_rate = FOOD_RATE_PRODUCTION)))
        _res = mn.main(food_rate_production = foodrate, plotting = False)
        results.append(np.array(_res[0]))
        foods.append(np.array(_res[1]))

# Linear fit of the data using SciPy
from scipy import stats
slope, intercept, r_value, p_value, std_err = stats.linregress(foods, results)
def predict_y_for(x):
    return slope * x + intercept
```

equilibrium.py

```
### Run the main.py file multiple times changing the food_rate_production parameters, and saves the ending food level and living blob Number  
### for further plotting and analysis
```

```
results = []  
foods = []  
food_rates_list = [10*i for i in range(1,20)]  
# In this way the print statements in bs.main() are suppressed  
with HiddenPrints():  
    for foodrate in tqdm(food_rates_list):  
        # results.append(np.array(bs.main(start_blob = NUM_BLOB, start_food = INITIAL_FOOD, food_rate = FOOD_RATE_PRODUCTION)))  
        _res = mn.main(food_rate_production = foodrate, plotting = False)  
        results.append(np.array(_res[0]))  
        foods.append(np.array(_res[1]))  
  
# Linear fit of the data using SciPy  
from scipy import stats  
slope, intercept, r_value, p_value, std_err = stats.linregress(foods, results)  
def predict_y_for(x):  
    return slope * x + intercept
```

Change the food_rate_production parameter

equilibrium.py

```
### Run the main.py file multiple times changing the food_rate_production parameters, and saves the ending food level and living blob Number
### for further plotting and analysis

results = []
foods = []
food_rates_list = [10*i for i in range(1,20)]
# In this way the print statements in bs.main() are suppressed
with HiddenPrints():
    for foodrate in tqdm(food_rates_list):
        # results.append(np.array(bs.main(start_blob = NUM_BLOB, start_food = INITIAL_FOOD, food_rate = FOOD_RATE_PRODUCTION)))
        _res = mn.main(food_rate_production = foodrate, plotting = False)
        results.append(np.array(_res[0]))
        foods.append(np.array(_res[1]))

# Linear fit of the data using SciPy
from scipy import stats
slope, intercept, r_value, p_value, std_err = stats.linregress(foods, results)
def predict_y_for(x):
    return slope * x + intercept
```

Change the food_rate_production parameter

Execute averages.py with varying parameter

equilibrium.py

```
### Run the main.py file multiple times changing the food_rate_production parameters, and saves the ending food level and living blob Number
### for further plotting and analysis

results = []
foods = []
food_rates_list = [10*i for i in range(1,20)]
# In this way the print statements in bs.main() are suppressed
with HiddenPrints():
    for foodrate in tqdm(food_rates_list):
        # results.append(np.array(bs.main(start_blob = NUM_BLOB, start_food = INITIAL_FOOD, food_rate = FOOD_RATE_PRODUCTION)))
        _res = mn.main(food_rate_production = foodrate, plotting = False)
        results.append(np.array(_res[0]))
        foods.append(np.array(_res[1]))

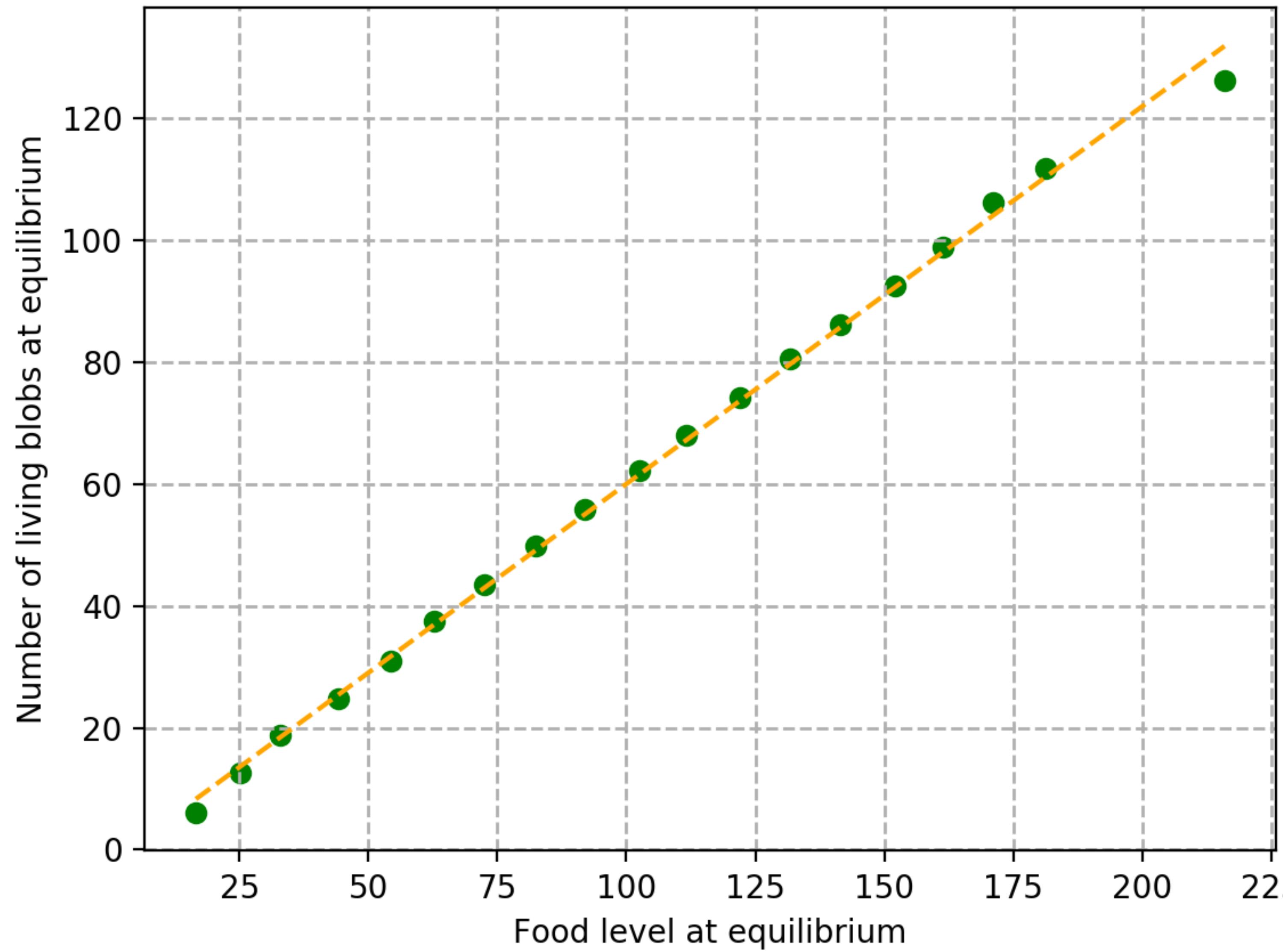
# Linear fit of the data using SciPy
from scipy import stats
slope, intercept, r_value, p_value, std_err = stats.linregress(foods, results)
def predict_y_for(x):
    return slope * x + intercept
```

Change the food_rate_production parameter

Execute averages.py with varying parameter

Using Scipy, fit with a linear regression of obtained results

Slope ~ 0.619 , Intercept ~ -1.935



Brief intuition

$$f = n_1 + 2n_2$$

f = food level

n_i = number of blobs eating i food

p_i = probability of a blob eating i food

N_{tot} = total number of living blobs

$$f = n_1 + 2(N - n_1)$$

$$f = 2N - n_1$$

$$f = 2N - Np_1$$

$$f = (2 - p_1) N$$

$$N_{tot} = n_1 + n_2$$

$$n_1 = Np_1$$

$$p_1 = \frac{(pt)^n}{n!} e^{-n} \Bigg|_{pt=1, n=1} = \frac{1}{e}$$

$$N = \frac{f}{(2 - p_1)} \Bigg|_{p_1=e^{-1}}$$

$$N \sim 0.61 f$$

So again...



So again...



maybe better
Object Oriented Programming (OOP)