Virtual Pet – Intro to Objects

## 

[Why use objects?](#_w7f048leib70)

[Overview of this resource](#_2yfhbepokl32)

[Assumed Knowledge](#_1o1dp6tva0qw)

[What is a class?](#_2o16ry7tuhzj)

[Manipulating the string method](#_p8mkn9iyo6yh)

[What is the self thing?](#_rjpxel9cekzq)

[Adding methods](#_rnkqr5u24ii6)

[Calling methods](#_92d32uzidqmy)

[Making our methods better.](#_dpz4rbq6asaf)

[Making a virtual pet main line](#_h8fgayel5od1)

[Extension 1: Let’s add some character art](#_l8h1oejvxk6i)

[Extension 2: Making multiple virtual pets](#_z3d7ankdylle)

[Glossary](#_defdjeaurc8)

[Appendix A: The Pet class](#_yq10ikk9d8uf)

[Appendix B: Updated methods for Pet with is\_dead()](#_x0oxwu7gx88a)

[Appendix C: Whole code single pet](#_4l7m8266712c)

[Appendix D: Whole code with character art](#_qmissev0gvn4)

[Appendix D: Whole code multiple pets](#_1cropz1foucw)

## 

## 

## Why use objects?

After programming had been around for a while people started to think about grouping information about a particular thing together. Then they thought about how to attach the behaviours of the thing along with the information about the thing. The result was Object Oriented Programming.

Have a look at the following example, what if we were making a game and we needed to keep track of a whole lot of information about the game character (let’s call him hero). We need to keep track of the coordinates, the image, the velocity, the number of lives, the score -- all about our hero. In the **imperative paradigm** we would need to do something like this…

| x = 20 y = 200 image\_location = "images/hero.jpg" velocity = 2.7 lives = 3 score = 245 |
| --- |

Then if we needed a “baddie” we’d have to store all the same information (pretty much) but for the baddie - so it’d start to get really messy... which data is about which thing?

| hero\_x = 20 hero\_y = 200 hero\_image\_location = "images/hero.jpg" hero\_velocity = 2.7 hero\_lives = 3 hero\_score = 245 baddie\_x = 20 baddie\_y = 200 baddie\_image\_location = "images/baddie.jpg" |
| --- |

The Object Oriented Paradigm allows us to have the data about a thing \*attached\* to the thing like this.

| hero = Hero() *#this makes a new instance of the class here* hero.x = 20 hero.y = 200  hero.image\_location = "images/hero.jpg"  *#notice the dot notation is used to refer to the variable that is \*attached\* to the hero.* |
| --- |

Note: you can see that not every variable belonging to the hero needs to be declared at the same time - we can have a bunch of values in the object given default values. This is awesome and useful.

## Overview of this resource

This resource will take you through the creation of a class for a virtual pet (similar to a tamagotchi, the 90s craze toy). We’re going to store a bunch of information about the pet including its name, age, image, hunger, boredom and sleepiness. At the end of the tutorial we will use our new Pet class to make multiple pets which we will care for. There is a link to a replit tutorial that you can fork and use to build your pet. <https://replit.com/@hogesonline/Virtual-Pet-Tutorial>

### Assumed Knowledge

In order to get the most out of this resource you need to have a familiarity with input, output, data types, loops, selections (if statements), and functions in Python.

## What is a class?

A class is like a template for an object. We will use it to set up the “shape” of the objects and then when we make objects we are making \*instances\* of the class. We define a class and then we need to give it variables. Giving a class variables is a little tricky, but hold on it’ll be explained.

| **class** **Pet**: |
| --- |

If you have seen object oriented programming before you might be tempted to just put the variables indented after the class name... kinda like this...

| **class** **Pet**:  *#don't do this!*  name = ""  age = 0  hunger = 5  boredom = 3  sleepiness = 3 |
| --- |

Unfortunately, those are class variables rather than object variables. We could use them to keep track of constants owned by the class e.g. number of Pets but actually we’ll steer clear of these for the time being. What we want instead is a set of attributes (variables) for each object. We do this with a **constructor.** A constructor is a special method (functions attached to a class are called methods) that **instantiates** a new object based on the class template. The trickiest part of this is the idea of “self”. Any object needs to know its own reference and all the things it actually owns. That’s where the concept of self comes from. It’s like saying, “Hey object! This is your stuff, we’re talking to YOU!”. Let’s have a look...

| **class** **Pet**:  **def** **\_\_init\_\_**(self):  self.name = ""  self.age = 0  self.hunger = 5  self.boredom = 3  self.sleepiness = 3  self.dead = **False** |
| --- |
|  |

Notice that these are all default values for the variables attached to the object.

Then we can use this Pet class like this:

| pet1 = Pet() *#note that you need the brackets* pet1.name = "Harry" print(f'This is your pet: {pet1.name}. It is {pet1.age} days old') |
| --- |

Even though it’s useful to have a bunch of these variables attached to the object to have default values - it would be nice to be able to name the Pet when we create it and not separately. We can do this by having a parameter passed to the **constructor.** Have a look at what that looks like:

| **class** **Pet**:  **def** **\_\_init\_\_**(self, pet\_name):  self.name = pet\_name  self.age = 0  self.hunger = 5  self.boredom = 3  self.sleepiness = 3 self.dead = **False** |
| --- |

Then we can use this Pet class like this:

pet1 = Pet("Harry") *#note that you put the pet\_name into the brackets*

## 

## Manipulating the string method

We want to print out our Pets! If you try printing your pet now you’ll hit a problem - Python prints a weird object address and that’s not very useful for humans...

| **class** **Pet**:  **def** **\_\_init\_\_**(self, pet\_name):  self.name = pet\_name  self.age = 0  self.hunger = 5  self.boredom = 3  self.sleepiness = 3  pet1 = Pet("Harry")  print(pet1) |
| --- |

This is the kind of thing you will see...

| <\_\_main\_\_.Pet object at 0x000001467D5B7790> |
| --- |

... Not very useful!

How about we change this so that we control what is printed? We could do this by writing a new method for our pet class that prints prettily. Have a look at this code:

| **class** **Pet**:  **def** **\_\_init\_\_**(self, pet\_name):  self.name = pet\_name  self.age = 0  self.hunger = 5  self.boredom = 3  self.sleepiness = 3  self.dead = **False**    **def** **pretty\_print**(self):  print(f"Name: {self.name}")  print(f"Age: {self.age}")  print(f"Hunger: {self.hunger}")  print(f"Boredom: {self.boredom}")  print(f"Sleepiness: {self.sleepiness}")  print(f"Death? {self.dead}")  print()  pet1 = Pet("Harry") pet1.pretty\_print() |
| --- |

This is one way to do it... We have created a new **method** called “pretty\_print” and it prints out each of the **attributes** of the Pet **class.** The issue here is that we have to call that method rather than just printing as we print everything else.

Handily there is another way!

We can use a **special method.** In Python, **special methods** are a set of predefined **methods** you can use to enrich your classes. They are easy to recognize because they start and end with double underscores, for example \_\_init\_\_ or \_\_str\_\_ .

Redefining the \_\_str\_\_ special method for your class allows you to change what is printed to screen using print() because you are defining the string representation of the class. Don’t forget you will need to return a string.

This is an example of **polymorphism.** This is one of the principles of OOP where we use the same methods to do different things depending on the context.

| **class** **Pet**:  **def** **\_\_init\_\_**(self, pet\_name):  self.name = pet\_name  self.age = 0  self.hunger = 5  self.boredom = 3  self.sleepiness = 3  self.dead = False    **def** **\_\_str\_\_**(self):  string = f"""Name: {self.name} Age: {self.age} Hunger: {self.hunger} Boredom: {self.boredom} Sleepiness: {self.sleepiness}  Dead? {self.dead}  """  **return** string  pet1 = Pet("Harry") print(pet1) |
| --- |

## 

## What is the self thing?

In Python self is used to represent the instance of the class (a.k.a an object). You might have seen this in other languages where they use this (like Java and Javascript - although there can be small differences in the meaning).

I think about self as making an object “self-aware” like saying to a baby “That is your hand, that is your foot.”. The method needs to know which object it’s acting on.

## Adding methods

We’ve already had a look at methods - but we might not understand what that term means. A method is a special kind of function that is “owned” or attached to a class and referenced by the object.

***Example:***

If we have created the class Pokemon which gets passed a nameand also a method called evolve(). We could do this:

| squirtle = Pokemon("Frank") squirtle.evolve() |
| --- |

A couple of things to notice here. While you can say Pokemon.evolve() this is a static method and not useful here - Pokemon all evolve individually this is not something the class should do. You could use Pokemon.evolve(squirtle) which is a bit unconventional, but perfectly correct.

The second thing to notice is that methods require the brackets. Remember our pretty\_print function was used like this pet1.prettyprint(). There can be parameters inside the brackets but whether there are or not - the brackets are required.

We need to add the following methods to our Virtual Pet:

1. Feed - which will reduce hunger by 3

2. Play - which will reduce boredom by 3

3. Sleep - which will reduce sleepiness by 5

4. Wait - which will increase age, and increase hunger, boredom and sleepiness

5. Is\_dead - which will check to see if the Pet has hit the thresholds we have set as the

maximum possible hunger, boredom and sleepiness

**Note about formatting:**

Methods need to be part of the class definition and so need to be indented to the right of the class name and lined up with the \_\_init\_\_ method in order to not get errors.

Let’s start with the feed() method. When we feed a Pet it should reduce its hunger by more than one because time passing will increase its hunger by 1.

What we need to do is reduce hunger by 3 and use a selection to make sure if hunger goes below zero it gets reset to 0 (we don’t want any negative numbers.)

| **def** **feed**(self):  *#check if the pet is dead, if it is, return nothing*  **if** self.dead:  **return**  *#reduce hunger by 3 but not below zero*  self.hunger = self.hunger - 3  **if** self.hunger < 0:  self.hunger = 0 |
| --- |

Now that you have followed along as we created the feed method. See if you can create the play, sleep, wait methods. (See *Appendix A* for solutions)

| **class** **Pet**:  **def** **\_\_init\_\_**(self, pet\_name):  self.name = pet\_name  self.age = 0  self.hunger = 5  self.boredom = 3  self.sleepiness = 3  self.dead = **False**    **def** **\_\_str\_\_**(self):  string = f"""Name: {self.name} Age: {str(self.age)} Hunger: {str(self.hunger)} Boredom: {str(self.boredom)} Sleepiness: {str(self.sleepiness)} Dead? {str(self.dead)} """  **return** string   **def** **feed**(self):  *#check if the pet is dead, if it is, return nothing*  **if** self.dead:  **return**   *#reduce hunger by 3 but not below zero*  self.hunger = self.hunger - 3  **if** self.hunger < 0:  self.hunger = 0   *#def play(self):*  *#def sleep(self):*  *#def wait(self):* pet1 = Pet("Harry") print(pet1) |
| --- |

Updating a value based on other values

We now have a way to make age, boredom, hunger, sleepiness go up and down based on some methods. Now we need to determine when the pet dies. We’re going to make a new method called check\_death() that manages this. These are the conditions I have chosen to use to determine if the pet should be dead. (Note: you can change these to make your pet harder or easier to keep alive)

* Boredom is at 10
* Sleepiness is at 10
* Hunger is at 10
* Age is at a random age between 15 and 20 or more

This is the start of the check\_death method... can you finish it off?

| **from** random **import** randint **def** **check\_death**(self):  **if** self.boredom >=10:  self.dead = **True**  *#add code for checking the other characteristics*  **if** self.age >= randint(15,20):  self.dead = **True** |
| --- |

Returning a value

We’re going to make a thing called an accessor (colloquially referred to as a “getter”). Generally we like to use “setters” (officially called mutators) and “getters” to access the variables stored in an object. We do this because of the principle of **encapsulation.**

**Encapsulation** is a way to protect object data and make objects more self-contained. In other words, encapsulation is like a protective shield for objects. It hides the object's internal details from the outside world, and only allows access to the object through its public methods. This helps to prevent accidental or malicious code from breaking the object.

Encapsulation also makes objects more self-sufficient. Because objects contain all of the data and methods they need to operate, they can be used in other parts of the program without having to worry about how they are implemented. This makes code more modular and reusable.

When we use a “getter” we are practicing **encapsulation**. Instead of checking whether our pet is alive or dead by using pet.dead we will use pet.is\_dead().So let’s write the method...

| **def** **is\_dead**(self):  *#just return the value stored in the dead attribute*  **return** self.dead |
| --- |

## Calling methods

We now have a Pet class and a bunch of methods that should enable us to make our pet work and do all the things we expect it to do. Before we write the main line we need to be sure we know how to call our methods properly.

If we had made a *function* called is\_dead we could call it this way:

| **if** is\_dead(): |
| --- |

Note the brackets! A *method* is a special kind of function so we still need the brackets but we need to include the object that we’re referring to.

| pet1 = Pet("Harry") if pet1.is\_dead(): |
| --- |

### Making our methods better.

Now that we have an is\_dead method we should go back and “fix” our feed, play, sleep and wait methods to ensure that we are following the principle of **encapsulation.** This is how we will need to change the feed() method. Go ahead and change the others (see Appendix 2 if you need help)

**def** **feed**(self):  
 *#check if the pet is dead, if it is, return nothing*  
 **if** self.is\_dead():  
 **return**

*#reduce hunger by 3*  
 self.hunger = self.hunger - 3

## 

## Making a virtual pet main line

We’re going to ask the user for a name for the pet - and make it. Then we will get an action to do with the pet over and over in a loop until the user types an empty loop. Each iteration of the loop we will need to print the pet so the user knows what the status is for boredom, sleepiness and hunger etc.

1. Ask the user for a name for the pet:

| name = input("What do you want your pet to be called?") pet = Pet(name) print(pet) |
| --- |

1. Ask the user for an action to do with the pet and while the action is not an empty string loop and print the pet in the loop. (It’s useful to print empty lines or dashes between each print of the pet.

| action = input("What do you want to do with your pet? ") **while** action != "":  *#put the if statement here*  print("-----------------------------------------------------")  print(pet)  print("-----------------------------------------------------")  action = input("What do you want to do with your pet? ") |
| --- |

1. Make a series of if statements to call our methods depending on what the action was that the user entered. Unless the user has entered something unexpected we always need to *wait* after each command.

| **if** action == "feed":  *#feed the pet*  pet.feed()  *#make time pass*  pet.wait()  *#fill in the if statement for the rest of the actions with elifs*  *#play*  *#sleep*  *#wait*  **else**:  print("You can only choose to feed, play, sleep or wait. ") |
| --- |

## Extension 1: Let’s add some character art

So now we have a pet, but it’s a bit boring to play with. It would be more fun if there was something cool to look at. This is your opportunity. We are going to design four character art drawings, one for a baby pet, one for an adolescent pet, one for an adult pet and one for a dead pet. We can also change the \_\_string\_\_ method so that instead of printing the number of hunger, sleepiness and boredom and age we will print the corresponding image and a visual indicator for each of the attributes.

Here are some examples:

| Name: Harry Pic:   W\_\_\_\_\_\_\_W  ᗧ(〇) W (〇)ᗤ  ᒪ/ \ᒧ  \\_\_\_\_\_\_\_\_\_/  ᗜ ᗜ  Hunger: ●●●●● Boredom: ●●● Sleepiness: ●●● |
| --- |

| ------------------------------------------- Name: Harry Pic:   V\_\_\_\_\_V  (O) w (O)  /| \_\_\_ |\  U U  Hunger: ●●●●●●●●●● Boredom: ●●● Sleepiness: ●●● |
| --- |

The big thing here is that we can use class variables to store our pictures.

Inside the Pet class we can add the following variables (in Python we use caps to mean “this variable is not going to change”)

The reason the formatting is weird is to make it so we can see the layout of the character art inside the code. The bit at the end [1:-1] strips off the first and last lines. Make three more constants for CHILD, ADULT and DEAD.

| **class** **Pet**:  BABY = r"""  v\_\_\_v  (o)\_(o)  `ʋ---ʋ`  """[1:-1] |
| --- |

We reference class variables this way: Pet.BABY

Now we need a new method that will return the pic based on the age of the Pet. We’re going to call it get\_pic(). The logic will be that if the Pet is younger than 3 it is a baby, if it is younger than 7 it is a child and otherwise it is an adult. It should show the dead picture if it is dead.

| **def** **get\_pic**(self):  *#check if the pet is dead*  **if** self.is\_dead():  **return** Pet.DEAD  *#check if it's still a baby*  **if** self.age < 3:  **return** Pet.BABY   *#complete the method for child and adult* |
| --- |

Now we need to change the \_\_str\_\_ **special method** so that instead of returning a string containing the age of the pet it returns the pic. See if you can do this and also change the Hunger, Bored and Sleepiness score to a visual indicator. (HINT: remember that you can multiply strings by an integer)

## Extension 2: Making multiple virtual pets

Because pets are objects based on a class we can actually have many of them at once. We can store them in a list or in a dictionary or even in another object. For this example we’re going to use a dictionary and use the name of the pet as a key.

The shape of this mainline will be like this:

1. Get input from the user
2. If the input is to “make blah” then check if a pet with that name already exists and if not, or if the pet is dead, create a new pet with that name
3. If the input is to “wait” then wait for all undead pets
4. If the input is to “play blah”, “feed blah”, “sleep blah” then check that the pet by that name exists and call the corresponding method on that object.
5. Loop through the dictionary of pets and call wait() for all of them that are alive
6. Print out all the pets in the dictionary
7. Stop when the user enters and empty string.

Here is some starter code:

| *#make an empty dictionary to store our pets* pets = {} action = input("What do you want to do? ") **while** action != "":  action = action.split()  **if** action[0] == "wait":  *#call wait on all of the pets in the dictionary pets*  el**if** action[0] == "create":  name = action[1]  **if** name **not** **in** pets **or** pets[name].is\_dead():  *#make a new pet and add to pets with the key name*  **else**:  *#message to user saying that they already have a pet called that*  *#complete the rest of the code for all the possible actions*  action = input("What do you want to do? ") |
| --- |

## 

## Glossary

*Attributes*: Data (or variables) that is associated with an object.

*Class*: A blueprint for creating objects. It defines the structure and behaviour of objects.

*Constructor*: A special method that is called when a new object is created. It is used to initialise the object's attributes.

*Data types*: A way to classify data so that the computer can understand how to use it. Common data types include integers, floats, strings, and booleans.

*Encapsulation*: A programming concept that bundles data and the methods that operate on that data into a single unit, called a class. This helps to protect the data from accidental or unauthorised modification, and makes objects more self-sufficient and independently functioning.

*Function*: A block of code that can be reused throughout a program. Functions can take input and return output.

*Imperative Paradigm:* The oldest and most basic programming approach. It is used in a wide variety of programming languages, including C, C++, Java, Python, and Ruby. Characteristics include using sequences, selections, iterations and functions and running from the top of the program to the bottom.

*Input*: Data that is entered into a program from the outside world. This can be done through a keyboard, mouse, or other device.

*Instantiate*: To create a new object from a class blueprint.

*List*: A data structure that stores a collection of items in a specific order.

*Loops*: A programming construct that allows you to repeat a block of code multiple times.

*Method*: A function that is defined within a class. Methods can access the class's private data and attributes.

*Object*: A self-contained entity that contains data and the methods that operate on that data. Objects are the building blocks of object-oriented programming (OOP).

*OOP (Object Oriented Programming)*: A programming paradigm that focuses on objects. OOP programs are made up of objects that interact with each other to perform tasks.

*Output*: Data that is produced by a program and sent to the outside world. This can be done through a display, printer, or other device.

*Polymorphism*: The ability of objects or methods to take on different forms. This allows objects or methods to be used in different ways without having to modify the code.

*Selection*: A programming construct that allows you to execute different code depending on the value of a condition. Also called an if statement.

*Self*: A reference to the current object. It is used to access the object's private data and attributes.

*Special method*: A method that is automatically called by the Python interpreter at certain times, such as when an object is created or destroyed.

## 

## Appendix A: The Pet class

| **class** **Pet**:  **def** **\_\_init\_\_**(self, pet\_name):  self.name = pet\_name  self.age = 0  self.hunger = 5  self.boredom = 3  self.sleepiness = 3  self.dead = **False**   **def** **\_\_str\_\_**(self):  string = f"""Name: {self.name} Age: {self.age} Hunger: {self.hunger} Boredom: {self.boredom} Sleepiness: {self.sleepiness} Dead? {self.dead} """  **return** string  **def** **feed**(self):  *#check if the pet is dead, if it is, return nothing*  **if** self.dead:  **return**  *#reduce hunger by 3 but not below zero*  self.hunger = self.hunger - 3  **if** self.hunger < 0:  self.hunger = 0    **def** **play**(self):  *#check if the pet is dead, if it is, return nothing*  **if** self.dead:  **return**  *#reduce hunger by 3 but not below zero*  self.boredom = self.boredom - 3  **if** self.boredom < 0:  self.boredom = 0    **def** **sleep**(self):  *#check if the pet is dead, if it is, return nothing*  **if** self.dead:  **return**  *#reduce sleepiness by 3 but not below zero*  self.sleepiness = self.sleepiness - 3  **if** self.sleepiness < 0:  self.sleepiness = 0    **def** **wait**(self):  *#check if the pet is dead, if it is, return nothing*  **if** self.dead:  **return**  *#increase age by 1*  self.age = self.age + 1  *#increase hunger by 1*  self.hunger = self.hunger + 1  *#increase sleepiness by 1*  self.sleepiness = self.sleepiness + 1  *#increase boredom by 1*  self.boredom = self.boredom + 1 |
| --- |

## Appendix B: Updated methods for Pet with is\_dead()

| **class** **Pet**:  **def** **\_\_init\_\_**(self, pet\_name):  self.name = pet\_name  self.age = 0  self.hunger = 5  self.boredom = 3  self.sleepiness = 3  self.dead = **False**   **def** **\_\_str\_\_**(self):  string = f"""Name: {self.name} Age: {self.age} Hunger: {self.hunger} Boredom: {self.boredom} Sleepiness: {self.sleepiness} Dead? {self.dead} """  **return** string  **def** **feed**(self):  *#check if the pet is dead, if it is, return nothing*  **if** self.dead:  **return**  *#reduce hunger by 3 but not below zero*  self.hunger = self.hunger - 3  **if** self.hunger < 0:  self.hunger = 0    **def** **is\_dead**(self):  **return** self.dead    **def** **play**(self):  *#check if the pet is dead, if it is, return nothing*  **if** self.is\_dead():  **return**  *#reduce hunger by 3 but not below zero*  self.boredom = self.boredom - 3  **if** self.boredom < 0:  self.boredom = 0    **def** **sleep**(self):  *#check if the pet is dead, if it is, return nothing*  **if** self.is\_dead():  **return**  *#reduce sleepiness by 3 but not below zero*  self.sleepiness = self.sleepiness - 3  **if** self.sleepiness < 0:  self.sleepiness = 0    **def** **wait**(self):  *#check if the pet is dead, if it is, return nothing*  **if** self.is\_dead():  **return**  *#increase age by 1*  self.age = self.age + 1  *#increase hunger by 1*  self.hunger = self.hunger + 1  *#increase sleepiness by 1*  self.sleepiness = self.sleepiness + 1  *#increase boredom by 1*  self.boredom = self.boredom + 1 |
| --- |

## Appendix C: Whole code single pet

This is the whole virtual pet code including the methods and mainline.

| **from** random **import** randint  **class** **Pet**:  **def** **\_\_init\_\_**(self, pet\_name):  self.name = pet\_name  self.age = 0  self.hunger = 5  self.boredom = 3  self.sleepiness = 3  self.dead = **False**   **def** **\_\_str\_\_**(self):  string = f"""Name: {self.name} Age: {self.age} Hunger: {self.hunger} Boredom: {self.boredom} Sleepiness: {self.sleepiness} Dead? {self.dead} """  **return** string  **def** **feed**(self):  *#check if the pet is dead, if it is, return nothing*  **if** self.dead:  **return**  *#reduce hunger by 3 but not below zero*  self.hunger = self.hunger - 3  **if** self.hunger < 0:  self.hunger = 0    **def** **is\_dead**(self):  **return** self.dead    **def** **play**(self):  *#check if the pet is dead, if it is, return nothing*  **if** self.is\_dead():  **return**  *#reduce hunger by 3 but not below zero*  self.boredom = self.boredom - 3  **if** self.boredom < 0:  self.boredom = 0    **def** **sleep**(self):  *#check if the pet is dead, if it is, return nothing*  **if** self.is\_dead():  **return**  *#reduce sleepiness by 3 but not below zero*  self.sleepiness = self.sleepiness - 3  **if** self.sleepiness < 0:  self.sleepiness = 0    **def** **wait**(self):  *#check if the pet is dead, if it is, return nothing*  **if** self.is\_dead():  **return**  *#increase age by 1*  self.age = self.age + 1  *#increase hunger by 1*  self.hunger = self.hunger + 1  *#increase sleepiness by 1*  self.sleepiness = self.sleepiness + 1  *#increase boredom by 1*  self.boredom = self.boredom + 1    **def** **check\_death**(self):  *#checks if the pet is dead*  **if** self.boredom >=10:  self.dead = **True**  **if** self.sleepiness >=10:  self.dead = **True**  **if** self.hunger >= 10:  self.dead = **True**  **if** self.age >= randint(15,20):  self.dead = **True**  *##### MAIN LINE ####* name = input("What do you want your pet to be called? ") pet = Pet(name) action = input("What do you want to do with your pet? ") **while** action != "":  *#put the if statement here*  **if** action == "feed":  pet.feed()  pet.wait()  **elif** action == "play":  pet.play()  pet.wait()  **elif** action == "sleep":  pet.sleep()  pet.wait()  **elif** action == "wait":  pet.wait()  **else**:  print("Invalid input")  *#We're not putting a wait here because the user made a mistake. We could if we wanted to be mean.*  pet.check\_death()  print(pet)  action = input("What do you want to do with your pet? ") |
| --- |

## 

## Appendix D: Whole code with character art

This is the whole virtual pet code including the methods with character art.

from random import randint

| **class** **Pet**:  #r is necessary so that Python doesn’t freak out with special characters  BABY = r"""  v\_\_\_v  (o)\_(o)  `ʋ---ʋ`  """[1:-1]  CHILD = r"""  V\_\_\_\_\_V  (O) w (O)  /| \_\_\_ |\  U U  """[1:-1]  ADULT = r"""  W\_\_\_\_\_\_\_W  ᗧ(〇) W (〇)ᗤ  ᒪ/ \ᒧ  \\_\_\_\_\_\_\_\_\_/  ᗜ ᗜ  """[1:-1]  DEAD = r"""  \_\_\_  /^ ^\  | R I P |  | |  \\| |//  """[1:-1]  **def** **\_\_init\_\_**(self, pet\_name):  self.name = pet\_name  self.age = 0  self.hunger = 5  self.boredom = 3  self.sleepiness = 3  self.dead = **False**   **def** **\_\_str\_\_**(self):  string = f"""Name: {self.name} Pic:  {self.get\_pic()} Hunger: {self.hunger \* "●"} Boredom: {self.boredom \* "●"} Sleepiness: {self.sleepiness \* "●"} """  **return** string  **def** **get\_pic**(self):  *#check if the pet is dead*  **if** self.is\_dead():  **return** Pet.DEAD  *#check if it's still a baby*  **if** self.age < 3:  **return** Pet.BABY   **if** self.age <= 5:  **return** Pet.CHILD   **if** self.age > 5:  **return** Pet.ADULT     **def** **feed**(self):  *#check if the pet is dead, if it is, return nothing*  **if** self.dead:  **return**  *#reduce hunger by 3 but not below zero*  self.hunger = self.hunger - 3  **if** self.hunger < 0:  self.hunger = 0    **def** **is\_dead**(self):  **return** self.dead    **def** **play**(self):  *#check if the pet is dead, if it is, return nothing*  **if** self.is\_dead():  **return**  *#reduce hunger by 3 but not below zero*  self.boredom = self.boredom - 3  **if** self.boredom < 0:  self.boredom = 0    **def** **sleep**(self):  *#check if the pet is dead, if it is, return nothing*  **if** self.is\_dead():  **return**  *#reduce sleepiness by 3 but not below zero*  self.sleepiness = self.sleepiness - 3  **if** self.sleepiness < 0:  self.sleepiness = 0  **def** **wait**(self):  *#check if the pet is dead, if it is, return nothing*  **if** self.is\_dead():  **return**  *#increase age by 1*  self.age = self.age + 1  *#increase hunger by 1*  self.hunger = self.hunger + 1  *#increase sleepiness by 1*  self.sleepiness = self.sleepiness + 1  *#increase boredom by 1*  self.boredom = self.boredom + 1    **def** **check\_death**(self):  *#checks if the pet is dead*  **if** self.boredom >=10:  self.dead = **True**  **if** self.sleepiness >=10:  self.dead = **True**  **if** self.hunger >= 10:  self.dead = **True**  **if** self.age >= randint(15,20):  self.dead = **True**  *##### MAIN LINE ####* name = input("What do you want your pet to be called? ") pet = Pet(name) action = input("What do you want to do with your pet? ") **while** action != "":  *#put the if statement here*  **if** action == "feed":  pet.feed()  pet.wait()  **elif** action == "play":  pet.play()  pet.wait()  **elif** action == "sleep":  pet.sleep()  pet.wait()  **elif** action == "wait":  pet.wait()  **else**:  print("Invalid input")  *#We're not putting a wait here because the user made a mistake. We could if we wanted to be mean.*  pet.check\_death()  print(pet)  action = input("What do you want to do with your pet? ") |
| --- |

## Appendix D: Whole code multiple pets

This is the whole virtual pet code including the methods with many pets. There have been many decisions implemented including when to wait to make time pass. Your might not match this exactly but try to think of all the edge cases for testing and run it to see what happens.

| **from** random **import** randint  **class** **Pet**:  *#r is necessary so that Python doesn't freak out with special characters*  BABY = r"""   v\_\_\_v  (o)\_(o)  `ʋ---ʋ`  """[1:-1]  CHILD = r"""  V\_\_\_\_\_V  (O) w (O)  /| \_\_\_ |\  U U  """[1:-1]  ADULT = r"""  W\_\_\_\_\_\_\_W  ᗧ(〇) W (〇)ᗤ  ᒪ/ \ᒧ  \\_\_\_\_\_\_\_\_\_/  ᗜ ᗜ  """[1:-1]  DEAD = r"""  \_\_\_  /^ ^\  | R I P |  | |  \\| |//  """[1:-1]     **def** **\_\_init\_\_**(self, pet\_name):  self.name = pet\_name  self.age = 0  self.hunger = 5  self.boredom = 3  self.sleepiness = 3  self.dead = **False**   **def** **\_\_str\_\_**(self):  string = f"""Name: {self.name} Pic:  {self.get\_pic()} Hunger: {self.hunger \* "●"} Boredom: {self.boredom \* "●"} Sleepiness: {self.sleepiness \* "●"} """  **return** string    **def** **get\_pic**(self):  *#check if the pet is dead*  **if** self.is\_dead():  **return** Pet.DEAD  *#check if it's still a baby*  **if** self.age < 3:  **return** Pet.BABY   **if** self.age <= 5:  **return** Pet.CHILD   **if** self.age > 5:  **return** Pet.ADULT     **def** **feed**(self):  *#check if the pet is dead, if it is, return nothing*  **if** self.dead:  **return**  *#reduce hunger by 3 but not below zero*  self.hunger = self.hunger - 3  **if** self.hunger < 0:  self.hunger = 0    **def** **is\_dead**(self):  **return** self.dead    **def** **play**(self):  *#check if the pet is dead, if it is, return nothing*  **if** self.is\_dead():  **return**  *#reduce hunger by 3 but not below zero*  self.boredom = self.boredom - 3  **if** self.boredom < 0:  self.boredom = 0    **def** **sleep**(self):  *#check if the pet is dead, if it is, return nothing*  **if** self.is\_dead():  **return**  *#reduce sleepiness by 3 but not below zero*  self.sleepiness = self.sleepiness - 3  **if** self.sleepiness < 0:  self.sleepiness = 0    **def** **wait**(self):  *#check if the pet is dead, if it is, return nothing*  **if** self.is\_dead():  **return**  *#increase age by 1*  self.age = self.age + 1  *#increase hunger by 1*  self.hunger = self.hunger + 1  *#increase sleepiness by 1*  self.sleepiness = self.sleepiness + 1  *#increase boredom by 1*  self.boredom = self.boredom + 1    **def** **check\_death**(self):  *#checks if the pet is dead*  **if** self.boredom >=10:  self.dead = **True**  **if** self.sleepiness >=10:  self.dead = **True**  **if** self.hunger >= 10:  self.dead = **True**  **if** self.age >= randint(15,20):  self.dead = **True**  *##### MAIN LINE ####* pets = {} action = input("What do you want to do? ") **while** action != "":  action = action.split()  **if** action[0] == "wait":  **for** pet **in** pets.values():  pet.wait()  **elif** len(action) == 2 **and** action[0] == "create":  name = action[1]  **if** name **not** **in** pets **or** pets[name].is\_dead():  **for** pet **in** pets.values():  pet.wait()  pets[name] = Pet(name)  **else**:  print("You already have a pet by that name")  **elif** len(action) == 2 **and** action[0] **in** ("play", "feed", "sleep"):  name = action[1]  **if** name **in** pets **and** **not** pets[name].is\_dead():  **if** action[0] == "play":  pets[name].play()  **elif** action[0] == "feed":  pets[name].feed()  **else**:  pets[name].sleep()  **for** pet **in** pets.values():  pet.wait()  **else**:  print("No pet by that name.")  **else**:  print("Invalid input, try again")  *#We're not putting a wait here because the user made a mistake. We could if we wanted to be mean.*  **for** pet **in** pets.values():  pet.check\_death()  print("---------------------")  print(pet)  print("---------------------")  action = input("What do you want to do with your pet? ") |
| --- |