Girls' Programming Network

Tamagotchi with Micro:Bits!

This project was created by GPN Australia for GPN sites all around Australia!

This workbook and related materials were created by tutors at:

Sydney, Melbourne and Perth

Girls' Programming Network

If you see any of the following tutors don't forget to thank them!!

Writers

Isabella Hogan
Alex Penna
Ashley Lamont

Testers

Part 0: Setting up

Intro to Micro:Bit

Task 0.1: micro:bits and pieces

Let's set up the micro:bit for programming today! You should have:

- **1 micro:bit chip**
- **1 USB cable**

1. Connect the small end of the USB cord to the middle port of the micro:bit
2. Connect the big end of the cord to your computer
3. Go to **python.microbit.org**

Task 0.2: Micro playground

First we're going to play around with the displays on **microbit.org** and test them on our micro:bits.

1. Make sure `from microbit import *` is at the top of your code.
2. Change the code under the `while True:` loop to display a duck and scroll your name instead using `display.show(Image.DUCK), sleep(1000)` and `display.scroll("Your name")`
3. Click the **'Send to micro:bit'** button, then follow the steps on the screen.
4. Try this out with other words and pictures.

Hint

Don't forget you have cheat sheets on the web page to help you code!
Remember to indent the code below the while loop!

☑ CHECKPOINT ☑

If you can tick all of these off you can go to Part 1:

☐ You are have connected your micro:bit to the computer

☐ You can display different pictures and words

Today's Project Plan - Tamagotchi

We're going to make a Tamagotchi electronic pet!
It will prompt the user to press A or B or the microbit logo to interact with the pet! Keep your pet happy and alive.

1. Start off the pet by showing and giving it a name and an image!
2. Show the pet's name, sleepiness, boredom and hunger.
3. Add the ability to play with, feed or send the pet to sleep.
4. Making the code a bit better
5. Make the ability for the pet to die, but keep it alive!
6. Watch your pet grow up.

+ more

Once your base pet works add cool extensions!

Part 1: Welcome to Tamagotchi!

Task 1.1: Name your file!

Now you've been introduced to your micro:bit, let's start working on the project!

1. At the top of the page, edit your project name to be 'tamagotchi'.
2. At the top of your code, use a comment to write your name

Task 1.2: Welcome

Let's welcome our user and get things set up!

Before the `while True:` loop, add the following:

1. Use `display.scroll` to say `"Welcome to Tamagotchi"`

2. Create some variables for our pet!
    a. `Name` (set this to whatever you want!)
    b. `hunger`, `boredom` and `sleepiness` should all be set to 0

☑ CHECKPOINT ☑

If you can tick all of these off you can go to Part 2:

☐ Your Micro:bit scrolls the welcome message!

☐ You've tried it on your physical Micro:bit

Part 2: Seeing our pet!

Task 2.1: Shake it like a polaroid picture

We want to display all of the current details about our pet if the Micro:Bit has been shaken. To do this let's first check if the board has been shaken

.

1. Delete what's already in the `while` loop
2. Inside the `while True:` loop, check to see if the accelerometer has detected the shake gesture
3. If it has, scroll what's currently in `name`

Hint: Using the Accelerometer

This is how you can test if the accelerometer (the inbuilt motion sensor) has detected a certain gesture.

```
if accelerometer.was_gesture("up"):
  display.scroll("Upwards!")
```

Task 2.2: What does our pet look like?

We now need to see what the pet actually looks like.

1. First, under where we are displaying the name, and still inside the `if` we want to display the image `DIAMOND_SMALL`
2. Then we need to `sleep` for 1 second

Hint: How to sleep?

Remember our sleep function works in milliseconds and one second is 1000 milliseconds so if we wanted to sleep for 5 seconds this is how we'd "**sleep**"

```
display.scroll("Z")
sleep(5000)
display.scroll("Z")
```

Task 2.3: How hungry, bored, and sleepy is our pet?

You should now scroll the values of each of our remaining variables

1. Still inside the `if` statement, scroll the text `"Hunger:"` and then the contents of our **`hunger`** variable then wait 50 milliseconds

2. Scroll the text `"Boredom:"` and then the contents of our **`boredom`** variable then wait 50 milliseconds

3. Scroll the text `"Sleepy:"` and then the contents of our **`sleepiness`** variable then wait 50 milliseconds

Hint: Changing variable data types

Remember our variables are numeric so to add them to text we must convert them to strings using the built-in function that converts to string

```
age = 15
display.scroll("Age is: " + str(age))
```

★ BONUS 2.4: Making your own photos!

We're currently using an inbuilt image to represent our pet but that's a little boring

Create and display your own image for your Tamagotchi - ask us for paper grids to design with, or use existing images from the micro:bit Image Cheat Sheet: http://bit.ly/images-microbit

You can program in your own images pixel by pixel like this:

```
gpn_heart = Image("09090:33903:90009:03030:00900")
display.show(gpn_heart)
```

☑ CHECKPOINT ☑

If you can tick all of these off you can go to Part 3:

- ☐ When your Micro:Bit is shaken, it displays the name
- ☐ It then displays an image for your pet
- ☐ It then scrolls to show the hunger, boredom, and sleepiness

Part 3: Interacting with our pet!

Task 3.1: Getting some Zzzs

We're going to flip our Micro:Bit over to send our Tamagotchi to bed.

1. Inside the `while True:` loop, make an `if` statement that checks if the accelerometer detects that the Micro:Bit is `"face down"`
2. Inside the `if` statement, reduce the **`sleepiness`** variable by 3 and show a sleepy face
3. We also only want to show this face for a couple seconds, so after you display the face, you'll need to use **`sleep`**, and then **`display.clear()`** to clear the screen

Task 3.2: Yummy yummy snacks!

We're going to make a new button to feed our Tamagotchi.

1. Inside our `while` loop, add a new `if` statement that checks if button **`A`** was pressed.
2. If **`A`** has been pressed, then decrease **`hunger`** by 3 and show a **`Happy`** face for a little bit

Hint: Detecting buttons

Here's a reminder on how to test if a button was pressed

```
if button_a.was_pressed():
  display.scroll("AAA")
```

Task 3.3: Play time!

Now, we're going to play some games with our Tamagotchi.

1. Inside our `while` loop (again), add a new `if` statement that checks if button **`B`** was pressed.
2. If **`B`** has been pressed, then decrease **`boredom`** by 3 and show a silly face for a little bit

☑ CHECKPOINT ☑

If you can tick all of these off you can go to Part 4:

- ☐ If you turn the Micro:bit over, it decreases the Tamagotchi's sleepiness and shows a sleepy face
- ☐ If you press the Micro:Bit's **A** button it decreases the Tamagotchi's hunger and shows a smiley face
- ☐ If you press the Micro:Bit's **B** button it decreases Tamagotchi's boredom and shows a silly face.

Part 4: Classes Rock!

Task 4.1: Getting things started

The first thing we'll need to do is to make a class called Pet and make an initialising function.

1. Make a `class` named Pet at the top of our code, underneath our imports
2. Inside the class make a function called `__init__` with a special 'self' parameter, and a parameter for the pet's name.
3. Copy all the variables we set up in part 1 (**`name, hunger, boredom and sleepiness`**) and paste it into the new `__init__` function (leave the old ones where they are as well for now)
4. Update all of them to have **`self.`** in front e.g. **`hunger`** becomes **`self.hunger`**
5. Update the **`name`** variable to use the **`name`** parameter instead

The next thing we'll need to do is make an object of that class to use in our game.

6. After where we display `"Welcome to Tamagotchi"`, make a new instance of the Pet class called **`my_pet`**. While making the object, pass in the name that you want your pet to have.

Hint: Making an __init__ method

A method is another name for a function that only works for a specific class. The **__init__** method is a special method that plans what the object is made up of when it is created.

In this example we see a class named rectangle that requires width and length as parameters for creation, and will store the width, length, and area as properties of the object upon creation.

Note: the self parameter is internal to the class object and is how it refers to itself. It must be included in the **__init__** definition parameters, but does not need to be included when creating objects of that class.

```
  class rectangle:
    # Making an __init__ method
    def __init__(self, width, length):
       self.width = width
       self.length = length
       self.area = self.width * self.length

  # Making a new rectangle object named rect
  rect = rect angle(8,10)
  print("The area of the rectangle is", str(rect.area))

>>> The area of the rectangle is 80
```

Hint: What's in a name?

There are some special names for methods and **__init__** is one of them. It has two underscores on either side of the word init to show python it's a special method and that it's being used to set up the class. You will need to remember this!

Hint: Making one version

With classes, they're basically like a template for code, setting up how it should act and what variables it should store, but we still need to make versions or "instances" to use them. This is how to make an instance of a class…

```
# Here's how you make an instance…
rect = rectangle(8,10)
print("The area of the rectangle is", str(rect.area))

>>> The area of the rectangle is 80
```

Task 4.2: Making our first custom method

Now we're going to start converting our current code into a bunch of methods for our classes. To do that let's start with one method so we can see how this needs to be done .

1. Create a function in the class called **`display`**. This should include self as a parameter when defining it, but should take in no parameters when calling it.

2. You will then need to copy all your code that you've already written to display your pet (anything after the `if` that tests to see if the Micro:Bit is shaken) into the function. you will need to then adjust any use of the original variables to now be referencing the class variables instead

In order to make sure that the method actually runs, you should then call it inside the if statement where you were previously displaying everything

Hint: Making a custom method

A method is another name for a function that only works for a specific class.

In this example we see a perimeter function that returns the results of a perimeter calculation.

Note how in both making the object and using its method we do not input a self parameter.

The self parameter is internal to the class object and is how it refers to itself. It must be included in the method definition parameters, but is not included when using the method of the class.

```
class rectangle:
  # Making an __init__ method (should be the first method)
  def __init__(self, width, length):
     self.width = width
     self.length = length
     self.area = self.width * self.length

  # Making a custom method
  def perimeter(self):
     return (self.width + self.length) * 2

# Making a rectangle object and using its perimeter method
rect = rectangle(8,10)
print("The perimeter is", str(rect.perimeter()))


>>> The perimeter is 36
```

Task 4.3: Again and Again and Again

You will now need to convert the rest of the functions so that everything is a method.

1. Following the process of the last step you will also need to make and convert the code to use the following three methods:
   a. sleep
   b. feed
   c. play

You can now also delete your old variables

☑ CHECKPOINT ☑

If you can tick all of these off you can go to Part 5:

- ☐ You should now have a new pet class
- ☐ It should have a display method that shows all the necessary information when the Micro:Bit has been shaken
- ☐ It should also feed, play, and sleep when the correct input is made

Part 5: Waiting and Dying

Task 5.1: Making a timer

Now we want to create a timer that starts when the code is run. It should reset every 10 seconds to mark the passing of one Tamagotchi 'year'.

1. Directly under the class, before any other code, make a variable called `start` and set it to the running time

2. Inside the `while` loop, check if the current running time is more than **10,000** milliseconds greater than `start` (this means that the difference between our stored run time - `start` - and whatever the current value is, should be **10,000**). If it is, reset `start` to the current running time.

3. For now you might want to quickly display something to show that it's working

Hint: running time built-in function

The microbit library has a built-in `running_time` function you can use to see the number of milliseconds since the micro:Bit was turned on

```
moment_in_time = running_time()
```

Task 5.2: Updating some attributes

Now we actually want to update all of our values after a “year” has passed.

1. Firstly, you need to make a new attribute in our **`__init__`** function called **`dead`** this should start as **`False`**

2. Secondly, make a method called **`wait`**.

3. Inside this method, write an **`if`** statement that checks whether **`self.dead`** is **`True`**. If it is, exit out of the function with the **`return`** keyword. This will stop the code from running any of the rest of the function.

4. You will then need to replicate this **`if`** statement at the top of each of your methods (except display). This will make it so that the pet's hunger/boredom/sleepiness won't change after it's died.

5. Inside the wait method after the **`if`** check, you will need to increase every value by 1 (That’s **`hunger`**, **`boredom`**, and **`sleepiness`**)

6. You will now need another **`if`** statement that checks if **`hunger`**, **`boredom`**, or **`sleepiness`** are above 10. If even one is, you will need to set the pet to dead.

7. Then, in your **`while`** loop, above where you are updating **`start`** you should call the **`wait`** function so that everything updates

Hint: Leaving a function with return statement

A function will end executing once it hits a `return` statement, regardless of what code is left beneath it.

```
class rectangle:
 # Making an __init__ method
 def __init__(self, width, length):
    self.width = width
    self.length = length
    self.area = self.width * self.length

 # Making a custom method
 def perimeter(self):
    # If width 0, exit using empty return statement
    if self.width == 0:
      return
    return (self.width + self.length) * 2

# Making a rectangle object and using its perimeter method
rect = rectangle(0,10)
print("The perimeter is", str(rect.perimeter()))

>>> The perimeter is None
```

Task 5.3: Are you dead?

We are now making a function that will tell us if the pet is dead.

1. Make a new method called **`is_dead`** and return whether **`dead`** is `True`

2. You will then need to go through all of your other methods and then any time you are checking whether **`dead`** is true you should use the method instead of the property.

☑ CHECKPOINT ☑

If you can tick all of these off you can go to Part 6:

- ☐ You now should have a timer that tracks a start point in time
- ☐ You now should have a wait method that checks your start point in time and updates it for each 'year' in your pets life
- ☐ It should also update all of the pet's attributes every time it updates the start variable, every 10 seconds
- ☐ The pet should be able to die if any of the pet's attributes values exceed 10

Part 6: We're growing up, up, up!

Task 6.1: How old is our pet?

We're going to start storing the age of our pet so we can know how long we have kept it alive for!

1. Inside your `__init__` function, make a new attribute called `age` that starts at 0
2. Inside your `wait` function, add one to `age` in the section you're updating all the other attributes

Task 6.2: What does our Tamagotchi look like?

We need to tell our code what our Tamagotchi should look like as it grows up!
In our class but **before** our `__init__` method, create four new class variables to hold images for our Tamagotchi:

1. Create a `baby_image` class variable that holds `Image.DIAMOND_SMALL`
2. Create a `child_image` class variable that holds `Image.SNAKE`
3. Create a `adult_image` class variable that holds `Image.BUTTERFLY`
4. Create a `dead_image` class variable that holds `Image.GHOST`

This will make them variables of the class itself that will never change between instances of the class

Task 6.3: Let's draw our tamagotchi

We need to draw our Tamagotchi as it grows up, to do this, we'll use a new method that draws it based on its current age.

1. Create a new method called `get_pic` in your class

2. In this method, return:
   - ***a.*** `dead_image` if the Tamagotchi is dead
   - ***b.*** `baby_image` if the Tamagotchi is alive and has `age` ≤ 3
   - ***c.*** `child_image` if the Tamagotchi is alive and has `age` > 3 and `age` ≤ 5
   - ***d.*** `adult_image` otherwise

3. Use this method to draw the current image for the Tamagotchi in our main draw method

Hint: Using "Class Variables"

Class variables are ones that apply to a class itself and not a single instance. This means it's slightly different when we want to use them. Instead of the normal `self.variable` for a class variable you have to do `name_of_class.variable`

for example:

```
class Rectangle:

  width = 5

  def __init__(self, length):
    self.length = length

  def area(self):
    return Rectangle.width * self.length
```

☑ CHECKPOINT ☑

If you can tick all of these off you can go to Extension 7:

- ☐ You initialise four image class variables before your **__init__** method
- ☐ You put an image into each of these, either a built-in image or one you've designed.
- ☐ You have a get_pic method that returns an image based on the Tamagotchi's age and dead-ness
- ☐ You draw the result of the get_pic method in your display method

Extension 7: Tamagotchi Friends!

We want our Tamagotchis to be able to talk to each other, and find friends! To do this, we'll use the radio to send and receive messages.

Task 7.1: Setting everything up

We're now going to set everything up.

1. At the top of your script, `import` the `radio` library, and add `radio.on()` below the import
2. In your `while` loop, add a new variable called `radio_msg` and set it to `radio.receieve()`
3. In our class `__init__` method, initialise a new variable called `loneliness` and set it to 0
4. Add our `loneliness` variable to the `display` and `wait` methods just like the existing `hunger/boredom/sleepiness` variables

Task 7.2: Being social

Now that we're storing the message we're getting let's actually work out what it says

1. Let's make a new method called `social`. Outside of the self parameter, it should take in one parameter ***`msg`***.
2. Inside our `social` method, do the following:
    a. Split the message by spaces, and then save the last item in the split message as a variable called `friend_name`
    b. Scroll `"Hi" + friend_name + "!"` on the display
    c. Decrease `loneliness` by 5
3. Inside your main `while True`, make another `if` statement that checks to see if ***`radio_msg`*** is not `None`. (This means we've received a message). If so, it should call the social method using the non-empty `radio_msg` variable.

Task 7.3: Talking!

Now we need to teach our Tamagotchi's to speak! Let's add a way to talk.

1. In our `while` loop, check to see if `pin0` is touched.

2. If `pin0` is touched, then we can use `radio.send` to send a greeting like `"Hi, I'm" + name` to any nearby Tamagotchi's. Remember to end your greeting with your name so other Tamagotchis can read it!

3. After sending a radio message, add a `sleep` for at least 1000 milliseconds, to avoid sending too many greetings at once!

Hint: Talking with friends

For this extension to work your tamagotchi will need another one to talk to so in order to test it on your physical one you'll need to wait till someone in the class has done this extension. For now you can test it online by scrolling down to the radio message section.

When you are touching the `pin 0` section of your microbit in reality, if it is not working make sure you hold the microbit by the other side of the bottom gold bar to to complete the touch circuit.

☑ CHECKPOINT ☑

If you can tick all of these off you can go to Extension 8:

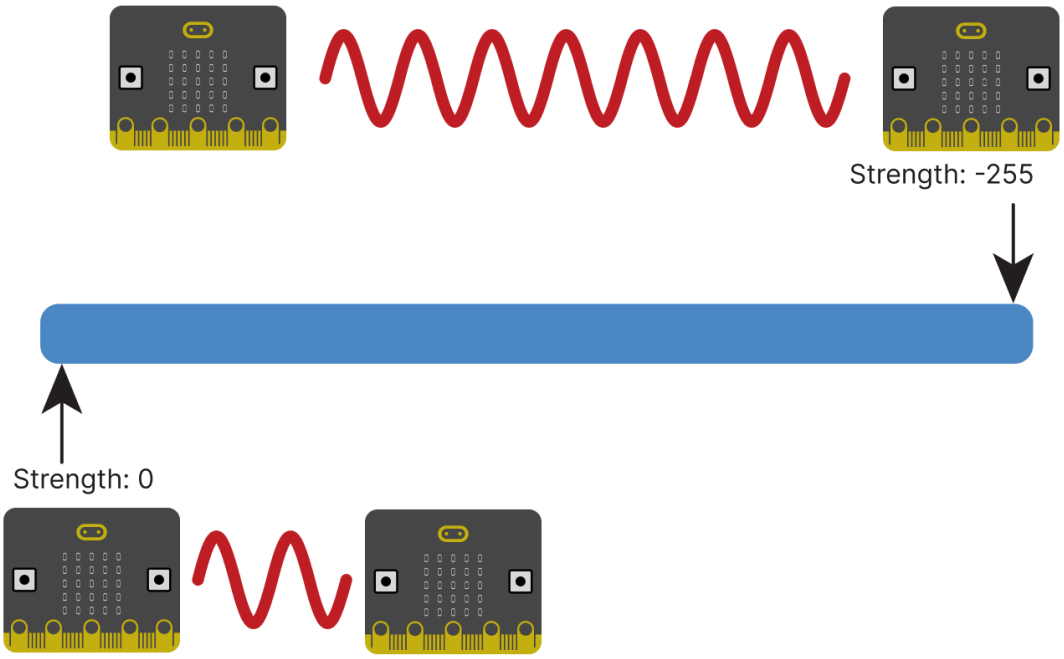
- ☐ Your tamagotchi should now be able to send a message to others if pin0 is touched
- ☐ It should also be able to receive messages from another tamagotchi.

Extension 8: Anti Social-Distancing (Tamagotchi friends pt. 2)

You must have done Extension 7 to do this!
Our Tamagotchi's can talk to each other, but they don't really care if they're actually together, or really far apart! We want them to be able to chat together when they're nearby and hanging out.

To do this, we can measure the signal strength of the signal that our radio gets!

Here's a diagram that shows our range of radio strength compared to distance. (The distances are not to scale but you could do some of your own testing to see what the actual range is if you're interested)



If we check the signal strength of messages that we get, we can compare it to the strength of a micro:bit that is kind of close to you. You can test this yourself and tweak the value, but we find that -35dBm is a good number!

Task 8.1: Changing our setup!

To start off we need to change how we're taking in our messages!

1. Delete where you define the `radio_msg` variable, and replace it with a new variable called `msg_details`. For `msg_details`, we'll save the value of `radio.recieve_full()` instead of `radio.receieve()`. This change will give us the message, signal strength, and timestamp of the message all together in a tuple!

2. You will also need to adjust the social method so that it takes in two parameters, a message and the signal strength

3. Adjust our `if` statement where we were checking if `radio_msg` isn't `None` so that it's it's checking `msg_details`

4. If it isn't `None`, make a new variable called `decoded_msg` and you will need to make it equal to `str(msg_details[0][3:], "utf8")`

Task 8.2: Different levels of social

Now we need to do different things with different signal strengths so that our tamagotchis are feeling more social the closer they are to the other tamagotchi.

1. Next, we can grab the signal strength as a second variable from `msg_details[1]` and store this in a variable called `strength`. We can then pass our `decoded_message`, along with the `strength` to `social`.

2. In `social`, before showing the friend's name, or decreasing loneliness, we should check if our signal strength is greater than -35 (or another number that works for you). Remember, a higher number (closer to 0) means a stronger signal!

3. If our message had a signal strength greater than our threshold, then we can greet it as normal and decrease loneliness.

4. If the signal was weaker than our threshold though, we should say `"I can hear " + friend_name` and decrease the loneliness by 3 instead of 5 like we do if we're close together.

Now, when friends greet you, you should see a different message depending on if your micro:bits are next to each other, or far apart!

☑ CHECKPOINT ☑

If you can tick all of these off you can go to Extension 9:

☐ Your micro:bit should respond differently to nearby friends, and further away friends

Extension 9: Many pets?

Right now you can only have one pet so let's rearrange some things so that you can have many!

Task 9.1: Setting up

We can get more pets by creating new instances of our Pet class

5. Firstly you need to make 2 or three pets with different names and add them to a list called **`pets`**.

6. Then, make a variable called **`current_pet`** and start it with the value 0 (this will store the pet we're currently looking at)

Hint: Lists

Remember this is what a list looks like…

```
my_favs = ["Chocolate", "Lollipops", "Skateboards", "Books"]
```

Task 9.2: Freeing up Button A and B

Now we need to free up `button A` and `button B` for cycling through our list of pets. We need to change the if statements that currently use them.

Let's assign our feeding action to a new trigger instead of `button A` press. Let's assign it to `pin1` touch, similar to our send a message section.

1. Where you're currently checking if `button A` was pressed, change it so instead it's checking if `pin1 is_touched`.
2. Add a delay of 1 second before you feed the pet so that we don't accidentally feed it too much.

For reassigning our play action, let's make the new trigger speaking to the pet! We can check the current sound value of the microphone to see if someone is speaking to the pet.

3. First, make a variable that stores the current amount of sound the microphone can hear so that we can use it as an input later. For this you'll need to make a variable called `sound_level` and in it, store `microphone.sound_level()`

4. Then, instead of where it's checking if `button b` is pressed, check if `sound_level` is greater than 180. This should fire if you talk to the microbit.
    a. You can mess around with the number here so that it requires different levels of noise to play with the microbit

Task 9.3: Scrolling through your pets

Now we want to make it so that we can access all of our pets to see, or interact with all of them.

1. Inside your `while` loop, check for every time you've referenced pet1 in order to interact with it, and instead use `pet[current_pet]` so that you're only working with the current pet

2. Now, make an `if` statement that checks id button a was pressed. If it was, decrease `current_pet` by 1 and then modulus it by the length of pets (this means you will cycle left through your pets when you press `button a`)

3. You will need to do the opposite if `button b` has been pressed. increase `current_pet` by 1 and modulus it by the length of pets

4. where you are currently making only the current pet wait, you should instead put a `for` loop that goes through each pet in pets and runs each of their wait methods so that they all update.

Hint: for loops

Remember this is what a for loop looks like…

```
nums = [1,2,3,4]
for num in nums:
  #do things
```

Hint: modulus function

Remember that a modulus is used to find the remainder when a number is divided by another number…

```
if 14%4 >0:
  print("Remainder =",14%4)

>> Remainder = 2
```

★ BONUS 9.4: Adding a new pet

Now we are making it so that you can add a new pet while the code is running.

1. First, you'll need to check if `pin1 is_touched()`. If it is, print `"creating new pet"` and ask the user for an input (to get the name of the new pet)

2. Then, you'll need to make a new instance of Pet with the name the user has given and add it to the list

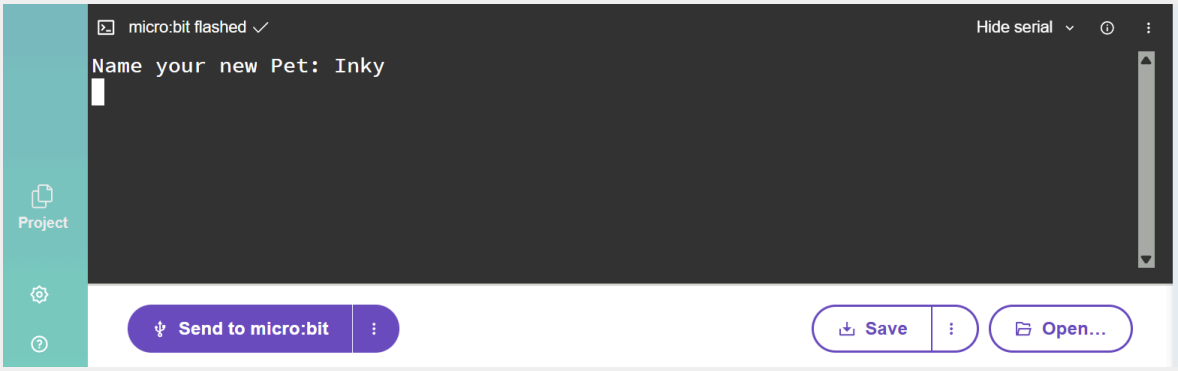
Hint: when you're running your code on your micro:bit, you'll need to make sure you use the send to microbit button and pair the computer with it so you can bring up the serial (this is where any prints or inputs from your physical microbit will show up)

Hint: Serial window for input to micro:Bit

When the micro:Bit asks for user input, you will have to expand the serial window at the bottom of your code in the python.microbit.org interface.

Click the Show serial ^ button to expand the serial window.

Expands to a window where you can give the micro:Bit your input



☑ CHECKPOINT ☑

If you can tick all of these off you're done:

- ☐ When you press A or B you can interact with different pets
- ☐ All interactions still work the same just under different inputs

Grids for designing your own micro:bit images