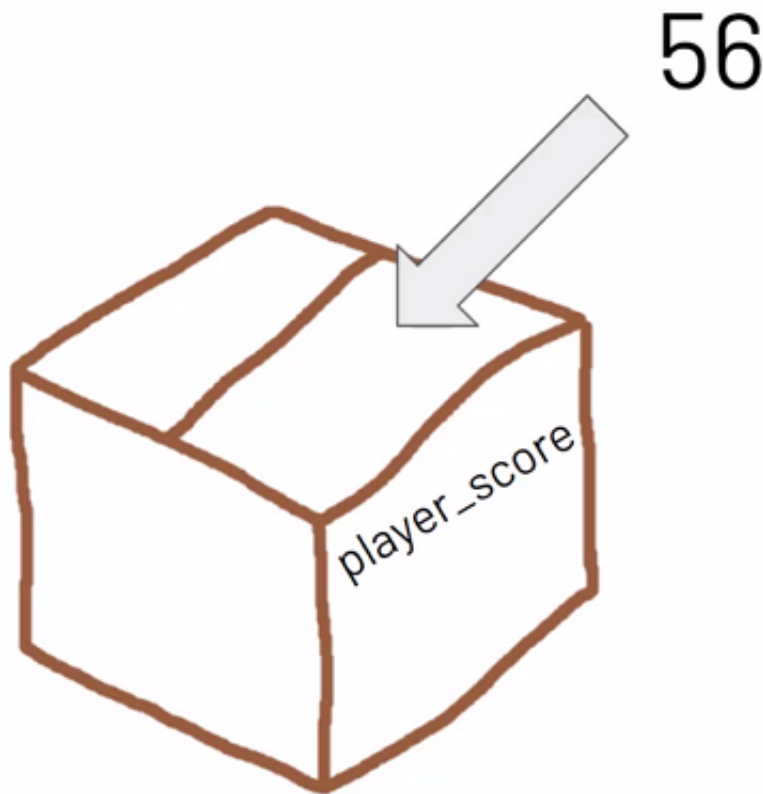




In this lesson, we're going to be looking at **variables** inside Python.

Variables constitute a very basic and important part of programming. Namely, they simply are chunks of data with a **name** and **value**. Variables are used whenever you want to store any piece of information or data in order to **read** it or **modify** it later on.

You can think of variables like a **box**. You can put a value such as a text or a number into the box and attribute a label to it to keep the data stored there:



There are *many* different types of data that we can store into a variable, also known as **data types**:

- **Integer** – whole numbers
- **Float** – decimal numbers
- **String** – texts
- **Boolean** – hold 'true' or 'false' values

Creating Variables in Python

Let's head over to a blank file in **Replit** to get started. Say we want to store the **age** of someone in a variable, so we do:

```
age = 20
```

We just need to write the name of the variable, the equal sign ('=') to attribute a value to it, and then the value itself. We read it as "age **equals** 20".



The above example is of an **integer** variable. To store a **decimal** number to keep track of the temperature, for instance, just type:

```
temperature = 25.39
```

As you notice, you can be very precise with floating-point numbers.

For **strings**, we need to enter the value in between **double quotation marks**:

```
country = "Australia"
```

Lastly, we can use **booleans** to check if it's **daytime** or nighttime:

```
is_day = True
```

Make sure to write boolean values with the first letter **capitalized** (i.e. **True** or **False**).

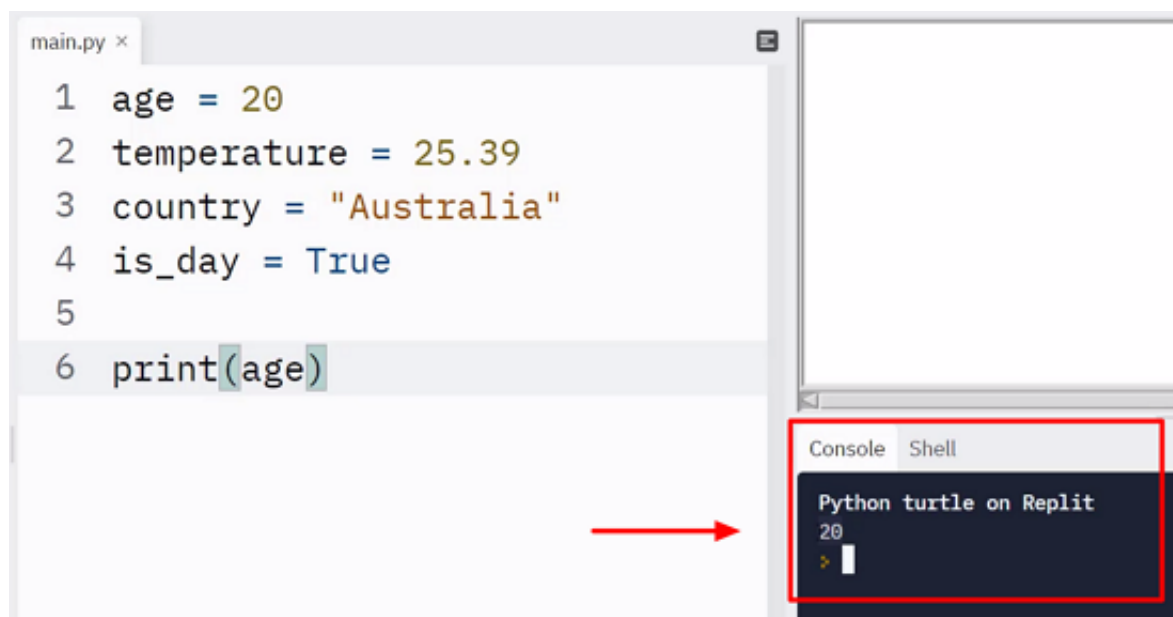
Print function

The print function allows us to **display messages** in the **console** we have at the bottom right of our screen. For it, **type** "print" and pass in the value to be printed between parentheses:

```
print(age)
```

The parameter of this function can be pretty much anything, varying from numbers to text, etc.

We can see the **result** of printing our 'age' variable in the console:





In this lesson, we'll learn how to modify variables in Python.

Variables can change during the execution of an algorithm. To see an example of that, let's **change** the value we have stored in our **country** variable as follows:

```
age = 20
temperature = 25.39
country = "Australia"
is_day = True

print(country)

country = "Ireland"

print(country)
```

By running it, we'll see that the **first** call of the print function printed out "**Australia**" while the second one printed "**Ireland**". As the code is run line by line from top to bottom, by the time we first printed the country variable we still hadn't modified it to "Ireland", and so the result was "Australia".

Note that if we print the country variable **before** we assign a value to it, we'll get an **error** since the computer doesn't know that this variable exists just yet:

```
print(country)

country = "Australia"
```

Modifying the Data Type of a Variable

Unlike other programming languages, Python allows us to **change** the **data type** of a **variable** throughout the program. If we were to attribute a boolean value or an **integer** to our **country** variable, we would successfully be able to do so:

```
age = 20
temperature = 25.39
country = "Australia"
is_day = True

print(country)

country = 900

print(country)
```

However, it's good practice **not** to alter the data type of the variables during the program as they are usually created to represent a **specific** thing, like temperature or age, and also to keep the code **organized** and **coherent**.



As a **challenge**, **create** the following **4 variables** that can be used to **describe** a **country** with the correct data types:

- country_name
- population
- landlocked
- border_size

By general formatting guidelines, we **substitute** the **space** in variable names with an **underscore** (as is the case of 'country_name' and 'border_size'). These variables stand for the name of the country, its population, whether or not the country is entirely surrounded by land, and its border length, respectively. Feel free to assign them any values you wish, from real life or not.

In the next lesson, we will go over the solution.



We hope you had a go at the challenge from last lesson. Below is the solution for the four variables we tasked you with setting up.

```
country_name = "Australia"  
population = 25000000  
landlocked = False  
border_size = 60000.256
```

For the `country_name` variable, we make it a **string**, given that it holds a text value, while `population` is an **integer**. `Landlocked` is a **boolean** and `border_size` is a **float** since it involves measuring kilometers/miles/meters - which are better represented with decimal numbers.

**Instructions:**

- Create variables to store information about a movie in a streaming app.
 - `movie_name` (string): Assign the name of the movie.
 - `release_year` (integer): Assign the year the movie was released.
 - `rating` (float): Assign a rating between 1.0 and 5.0 stars.
 - `is_free` (boolean): Assign `True` if the movie is available on the free plan, otherwise `False`.
- Use the `print()` function to display each piece of information on the screen.



In this lesson, we're looking at operators in Python.

Operators are special symbols in Python that carry out **arithmetic** or logical computation, such as the operators for addition, subtraction, multiplication, and division that we're going to see below.

Let's begin by having a '**health**' variable for a game, for example, where the player's health starts at 100:

```
health = 100

health = health - 20
print(health)
```

If the player takes damage, we **subtract** that damage from its health. For that, we're **assigning** to the health variable a **new** value which is what's **currently** stored in health **minus** 20.

As a challenge, try adding health to the variable instead. We'll go over this in the next lesson, as well as discuss multiplication and division.



In this lesson, we'll continue our discussion on operators.

First, let's go over our challenge solution. If the player gets **healed**, we then need to **add** to its health instead:

```
health = health + 50
print(health)
```

To **multiply**, we use the **asterisk** symbol as follows:

```
health = health * 1.5
```

Finally, to **divide** we use the **forward-slash** operator:

```
health = health / 2
```

Alternatively, we can apply the operators directly to other **variables** as well. So, for our first example, we could declare a variable called **damage** and subtract it from health:

```
health = 100
damage = 90

health = health - damage
print(health)
```

As damage is equal to 90, the console will print out a result of **10** in this case.



Operators

So far we've looked at the addition (+), subtraction (-), multiplication (*) and division (/) operators. Here's how we can create a variable, then increase the value by 10:

```
num = 5
num = num + 10
```

When writing code though, programmers like to shorten things. Here's how we can shorten the command: *num = num + 10*.

```
num = 5
num += 10
```

+= is the combination of an assignment and addition operator. In the example above, we create a variable called *num* and set it to 5. Then we increase the variable's value by 10.

We can also shorten subtraction, multiplication, and division.

```
num = 10

# increase num by 5
num += 5

# subtract 3 from num
num -= 3

# multiply num by 1.5
num *= 1.5

# divide num by 2
num /= 2
```



In this lesson, we'll start to learn about conditions in Python.

Conditions allow us to check if a **statement** is **true** or **false** and execute code based on that outcome. This mechanism makes it possible for us to branch out our program as some blocks of code will only be run if a certain condition is met.

If statements

Let's declare **2 variables** and print out a message **only** if their values are **equal**:

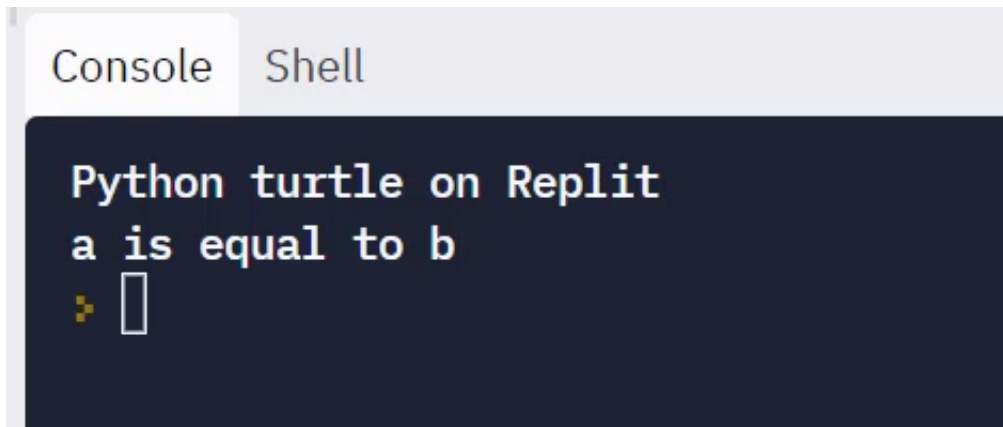
```
a = 5
b = 8

if a == b:
    print("a is equal to b")
```

To do this check, we type the keyword **if** then add the **condition** that'll return to us a true or false value. The condition we're interested in verifying is whether a is equal to b and for that, we use the **comparison** operator **==** in Python. After the condition is entered, we add a **colon** symbol and go to a new line. You'll see the cursor has been automatically **indented** or you can do it manually by pressing Tab.

Note: Python uses **4 spaces** as indentation by default. However, this number can vary, although a minimum of 1 space is required in order for the computer to be able to **distinguish** between the lines of code that are part of the if block and the ones that aren't.

Running our code will as is not print anything out in the console since a is **not** equal to b. If we **change b** to **5**, we'll then see our **message** in the console:





In this lesson, we'll continue our study on **conditions** in Python.

Apart from checking if two variables are equal to one another, we can also check if one is **greater** than the other using **if** statements:

```
a = 5
b = 10

if a > b:
    print("a is greater than b")
```

This if condition reads “**Is 5 greater than 10?**” and uses the **>** symbol for greater than. If it's **true**, the print will be **executed**. If it's not, nothing is going to happen.

Similarly, we can also check if a is **less than** b. Go ahead and try this out as a challenge, and we'll go over the solution in the next lesson.



In this lesson, we'll go over our conditions challenge from the last lesson.

For our challenge, to check if a is **less** than b we use the **<** operator:

```
a = 1
b = 10

if a > b:
    print("a is greater than b")

if a < b:
    print("a is less than b")
```

It's important to notice that you **don't** want to be **inside** the previous if block indentation here, so always remember to check that your code is **correctly indented**.

Noticeably, there are a lot more conditional operators than the ones we've seen here but we won't be going over those for this course.

Conditions

Conditions are a vital part in creating dynamic algorithms. In the video lessons we went over **if statements**. But that's not all there is! Along with if statements, we also have *else if* and *else* statements. These go side by side with if statements and allow for additional condition checks if an initial if statement was false.

Else

Confusing? Let's start simply with **else**. As an example, let's have an algorithm which has a name variable. We want to print out a message if that name is *Jack*, and another message if the name is not *Jack*.

```
name = "Jack"

if name == "Jack":
    print("You are Jack")

if name != "Jack":
    print("You are not Jack")
```

Now that will work, one of the two messages will print out depending on the name variable's value.

A better way to write this algorithm though, is with an *else* statement.

```
name = "Jack"

if name == "Jack":
    print("You are Jack")
else:
    print("You are not Jack")
```

Else is what we can define after an if statement. If the *if statement* is false, then the code under *else* will execute.

Else If

Along with *if* and *else*, we have **else if**. This is like a combination of *if* and *else*. Else if allows us to stack if statements. For this, we use the **elif** keyword.

```
name = "Bob"

if name == "Jack":
    print("You are Jack")
elif name == "Bob":
    print("You are Bob")
else:
    print("You are not Jack or Bob")
```



So in this example, we are checking if the name is *Jack*. If not, then we go down to the *elif* statement to see if the name is *Bob*. If even that condition is false, then the *else* statement code will run.

Yet if *name == "Bob"* is true, neither the *if* or *else* statement code will run. Think of it like this:

- If *this* is true, do this. Otherwise, if *this* is true, do this. If none of those things were *true*, then do this.

We won't be using *elif* or *else* throughout the rest of the course, yet it's important to know these moving forward with programming.

Imagine you are creating a simple medical program to check if a patient's temperature indicates a fever.

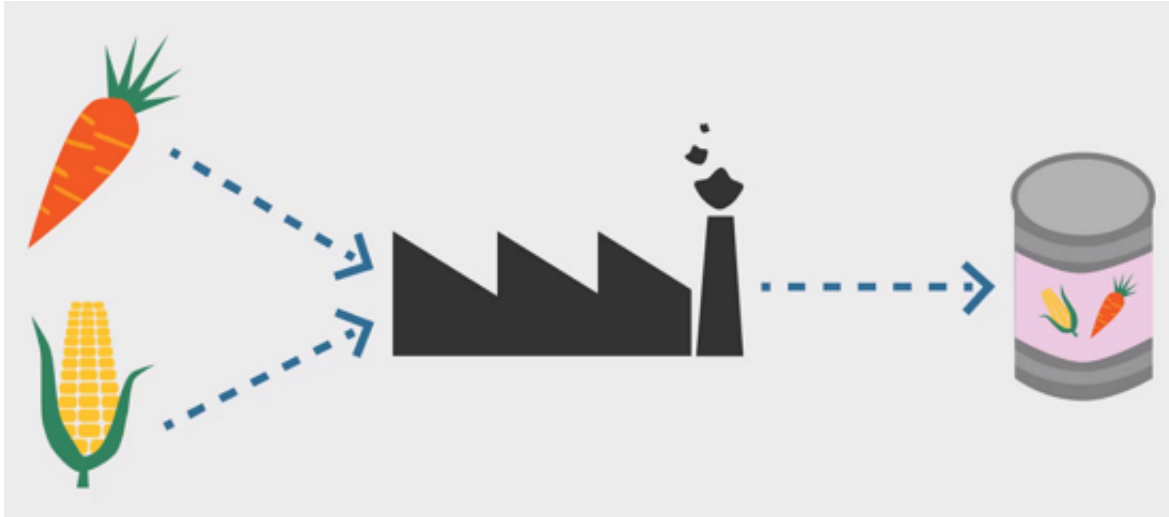
Instructions:

- Create a variable called `patient_temperature` and assign it a value in degrees Celsius (e.g., 38).
- Write an if statement to check if `patient_temperature` is greater than 37.
 - If it is, use `print()` to display "Alert: Patient has a fever."
 - If not, use `print()` to display "Patient's temperature is normal."

In this lesson, we'll be looking into functions in Python.

Functions are essentially blocks of **reusable code**, that is, a piece of code that we can run as many times as we want.

We can think of functions as a factory, when given certain **inputs** it'll **produce** the **output** accordingly:



Let's say we have the following code:

```
print("Hello Bob")
print("Hello Steve")
print("Hello Mary")
```

If we wanted to **change** the phrase by adding an exclamation mark at the end, we would have to go through **each** line manually adding it. As you can imagine, this process becomes quite impractical to do for large pieces of code. However, we can simplify all that with functions.

To **define** a function we type the **def** keyword, then the **name** of the function, and we can pass in the **parameters** or arguments for this function in between parentheses:

```
def say_hello(name):
    print("Hello " + name)
```

Inside our function, we want to call the **print** function passing in the concatenation of "Hello" and the name sent as parameter.

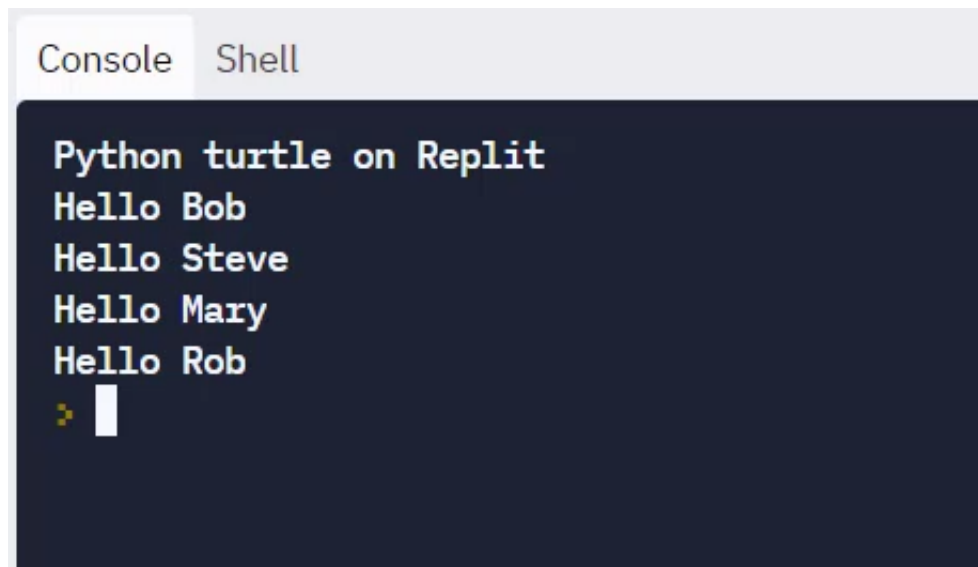
If you try to run it, nothing's going to happen because our say_hello function has only been declared, but it isn't being called. To **call** it, go to a new line and write down the name of the function alongside the parameter you want to send over:

```
def say_hello(name):
    print("Hello " + name)
```




```
say_hello("Bob")  
say_hello("Steve")  
say_hello("Mary")  
say_hello("Rob")
```

Here we're calling our function several times, one for each different name that we want to print out to the console:



The screenshot shows a Replit console window with a dark background. At the top, there are two tabs: 'Console' and 'Shell'. The 'Console' tab is selected. The output in the console is as follows:

```
Python turtle on Replit  
Hello Bob  
Hello Steve  
Hello Mary  
Hello Rob  
> |
```

Now, if we need to **alter** the phrase being printed, we do it only **once** directly where we're defining our function:

```
def say_hello(name):  
    print("How are you " + name)  
  
say_hello("Bob")  
say_hello("Steve")  
say_hello("Mary")  
say_hello("Rob")
```

And we see that the changes have taken place correctly:



Console

Shell

```
Python turtle on Replit
```

```
How are you Bob
```

```
How are you Steve
```

```
How are you Mary
```

```
How are you Rob
```

```
> 
```



In this lesson, we're continuing with functions in Python.

Let's **import** Python **Turtle** to have a function that **moves** us a set distance and then **rotates** a certain number of degrees:

```
from turtle import *

def move_and_turn(distance, angle):
    forward(distance)
    right(angle)
```

Note that not all functions have parameters, and if they don't you can just leave the parentheses empty.

Let's then **call** our function passing in the actual values for the parameters we've established:

```
from turtle import *

def move_and_turn(distance, angle):
    forward(distance)
    right(angle)

move_and_turn(100, 45)
move_and_turn(50, 90)
```

Running it, we have:



Remember to always **define** the function **before** you call it, otherwise you'll have an **error** as the function doesn't exist for the computer just yet.

In fact, all **forward**, **right**, or **print** are functions (basically whenever you have a word and parentheses, that is a function). The forward and right functions exist inside the **Turtle** library that we're **importing** into our code and we simply need to **call** them to use their functionalities. Also, print is a Python function that receives a string as a parameter and prints it out in the console.



You'll definitely be using a lot of functions whenever you're coding more complex algorithms as they play a crucial role in programming.

Functions

For our upcoming projects, we'll be using functions. They are a vital aspect of programming and developing algorithms. So in this text lesson, we're going to explore functions a bit more.

Let's cover the *structure* of a function first.

We start a function by writing the **def** keyword. Then we give the function a name. In brackets we can then define some parameters (these are optional). Parameters are basically inputs into an algorithm.

For example, if we have a function where we want to add two numbers together and print out the result, it would look like this:

```
def add_and_print (a, b):  
    sum = a + b  
    print(sum)
```

Returning Values

Along with inputs into a function, we can also have **outputs**. So let's use the function above, but this time it will just add two numbers together and return the result.

```
def add_numbers (a, b):  
    sum = a + b  
    return sum
```

The **return** keyword will end the function and return a value. In our case the sum of *a* and *b*.

We can then use this function like so:

```
def add_numbers (a, b):  
    sum = a + b  
    return sum  
  
num = add_numbers(150, 2063)
```

Instructions:

Create a program that defines a function to draw a square using Python Turtle. The function should:

- Be named `draw_square`.
- Take one parameter for the size of the square.
- Use the turtle to draw a square of that size.

Tips:

- Use the `forward()` and `right()` Turtle functions to draw sides and turn.
- You can move the turtle to different positions using `penup()`, `goto(x, y)`, and `pendown()`.
- Remember to define your function before you call it.
- Be creative with the placement and sizes of your squares.

Optional extra challenges:

- Add a second parameter that takes the color of the square. The square should be filled with that color.
- Move the turtle after drawing the square and draw another square with a different size. Repeat this multiple times to draw multiple squares of different colors.



In this lesson, we're going to start our Balloon Popper project with Python Turtle.

We'll go through **4 steps** of our **project development** in the next lessons:

1. Definition
2. Design
3. Implementation
4. Evaluation

Definition

Here we're defining the **criteria** for our project, which are:

- **Detect keyboard inputs** to have the computer increase the balloon size and pop it.
- **Require multiple inputs** to pop the balloon, as we need to have a process of inflating the balloon until it gets sufficient size to pop.
- **Utilize variables, conditions, and functions** as learned in our previous lessons.

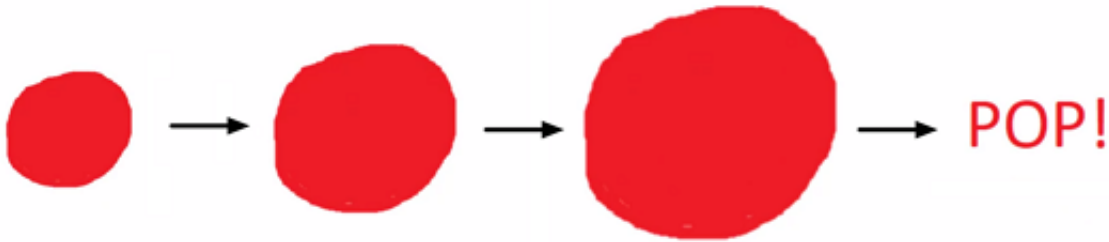
In the next lesson, we'll do the designing step of our project.

In this lesson, we're doing the design of our Balloon Popper project.

Diagram

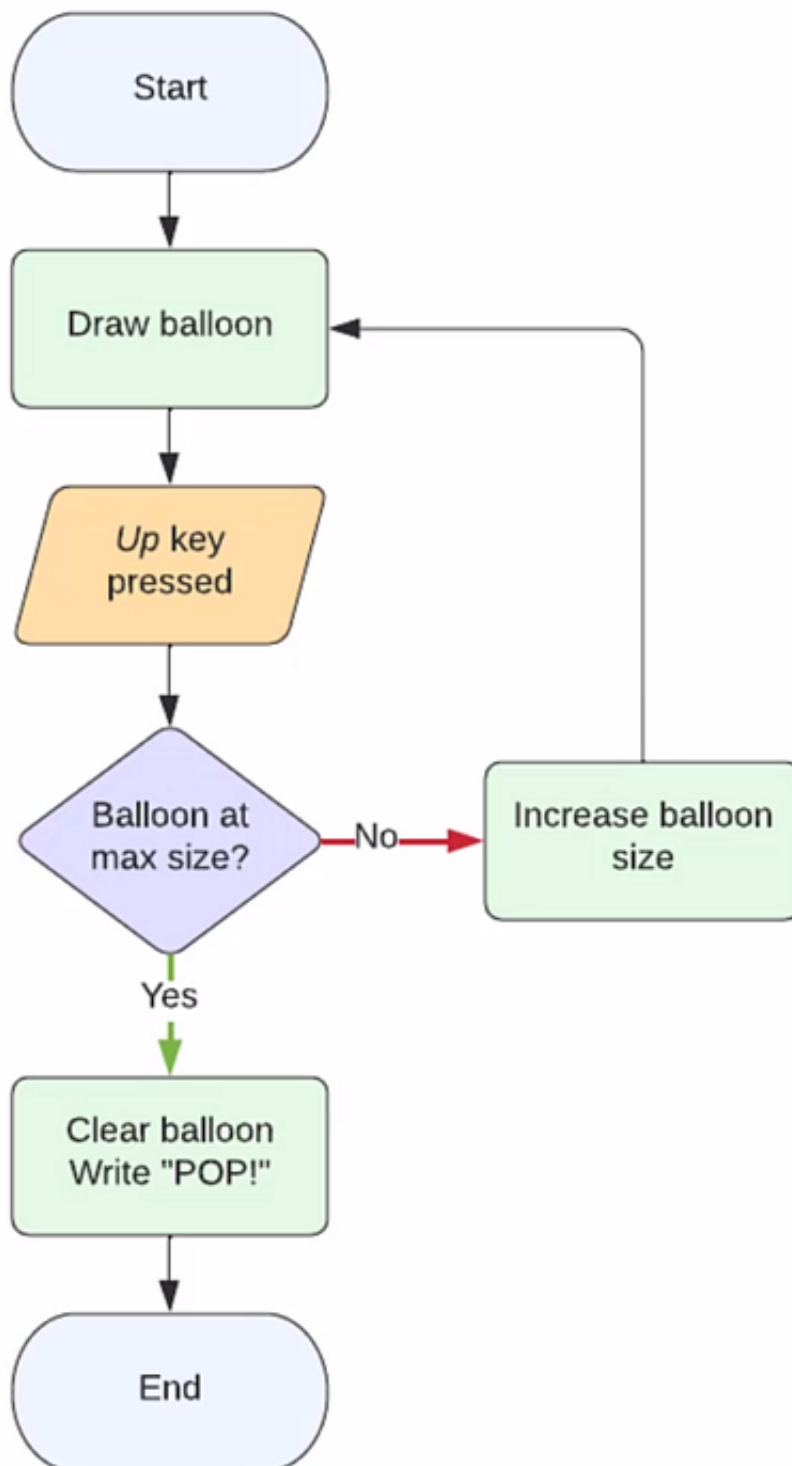
We'll start off with a diagram to better visualize what we're creating.

The idea of our algorithm is fairly simple. We're going to have a **red dot** drawn on the screen which will **increase** in size every time the **up arrow key** on the keyboard is pressed. Once it reaches a certain size, it'll **pop** leaving behind a message for the player:



Flowchart

Next, we're going to make a flowchart to denote the flow of our algorithm:



First, we **draw** the balloon before we start getting the player's input on it. Then we'll check for the up arrow key to be pressed. Whenever that happens, we're **checking** if the balloon is at its maximum size. If the balloon is ready to pop, we clear the balloon from the screen and write "POP!" In case the balloon is **not** at its maximum size, we're going to **increase** the size of the balloon and then draw it again. We'll repeat that process until the balloon is big enough to pop.



Pseudocode

Our pseudocode follows along the same lines of the flowchart above:

START

Draw the balloon

Has the key been pressed?

 Has the balloon reached max size?

 Clear the balloon

 Write "POP!"

 If not

 Increase balloon size

 Draw the balloon

END

We start by **drawing** the balloon, then we check if the up **key** has been pressed. If so and the balloon has reached its **maximum** size, we clear the balloon and write **pop** on the screen. If not, we **increase** the balloon size and **redraw** it until it is ready to pop.

In this lesson, we're starting the implementation of our Balloon Popper project.

The first thing we're doing is to **import Turtle** and **set** the 2 **variables** we're going to need:

```
from turtle import *  
  
diameter = 40  
pop_diameter = 100
```

The **diameter** variable has the diameter of the balloon when **initially** created in terms of **pixels**, and **pop_diameter** defines the diameter the balloon has to have to **pop**.

We'll also have 2 functions for this project, let's begin by implementing the **draw_balloon** first:

```
from turtle import *  
  
diameter = 40  
pop_diameter = 100  
  
def draw_balloon():  
    color("red")  
    dot(diameter)  
  
draw_balloon()
```

In this function, we're **setting** the **color** of the turtle to be red and then, instead of manually drawing a circle and filling it in (with the `begin_fill` and `end_fill` commands), we're using the **dot** function (passing in the **diameter** we've specified) as it already does that for us.

Note that the circle function receives the radius of the circle (which is half the size of the diameter) as a parameter while the dot function takes the full diameter directly.

Next, we **call** our function to test if it's working properly and there's our **initial red balloon** on the screen:





If you're not using Replit and are having trouble running your code with your chosen IDE, try adding turtle's *done()* command at the very end of your code:

```
from turtle import *

diameter = 40
pop_diameter = 100

def draw_balloon():
    color("red")
    dot(diameter)

def inflate_balloon():
    global diameter
    diameter = diameter + 10
    draw_balloon()

    if diameter >= pop_diameter:
        clear()
        diameter = 40
        write("POP!")

draw_balloon()

onkey(inflate_balloon, "Up")
listen()
done()
```

This will keep the program window open normally and it should run just fine.

In this lesson, we're continuing our code for the Balloon Popper.

Let's start **implementing** the **second** function we're going to need for this project, the **inflate_balloon**:

```
from turtle import *

diameter = 40
pop_diameter = 100

def draw_balloon():
    color("red")
    dot(diameter)

def inflate_balloon():
    diameter = diameter + 10
    draw_balloon()

draw_balloon()
inflate_balloon()
```

Here we're **increasing** the **diameter** of our balloon by a small amount (in this case, 10) and then **redrawing** it bigger on the screen.

Scope of Variables

Python and so many other programming languages work with the idea of **scope**. Namely, what's inside of a function (i.e. code indented to the function) can't be accessed from outside of the function just as what's outside can't be accessed from within the function. The variables created **inside** the scope of a **function** are called **local variables**. On the other hand, variables declared **outside** the scope of a function are **global** variables.

If we run our code now, we'll get an error since we **can't** access the diameter variable that's at the beginning of our code from inside our **inflate_balloon** function and we also haven't defined a local diameter variable inside our function either. As we need to **access** the **global diameter** variable, we have to specify it in our function by adding 'global diameter' before assigning the updated value to it:

```
from turtle import *

diameter = 40
pop_diameter = 100

def draw_balloon():
    color("red")
    dot(diameter)

def inflate_balloon():
    global diameter
    diameter = diameter + 10
    draw_balloon()

draw_balloon()
```



```
inflate_balloon()
```

If we try running it again, it shows us the first balloon we've drawn and straight away increases its size a tiny bit.



The code in the video for this particular lesson has been updated, please see the lesson notes below for the corrected code.

In this lesson, we're going to set up our **inflate_balloon** function so that it gets called whenever the **up** arrow key is **pressed**.

For that, we're going to **replace** the call we were making at the end of our code to call the **onkey** function instead:

```
from turtle import *

diameter = 40
pop_diameter = 100

def draw_balloon():
    color("red")
    dot(diameter)

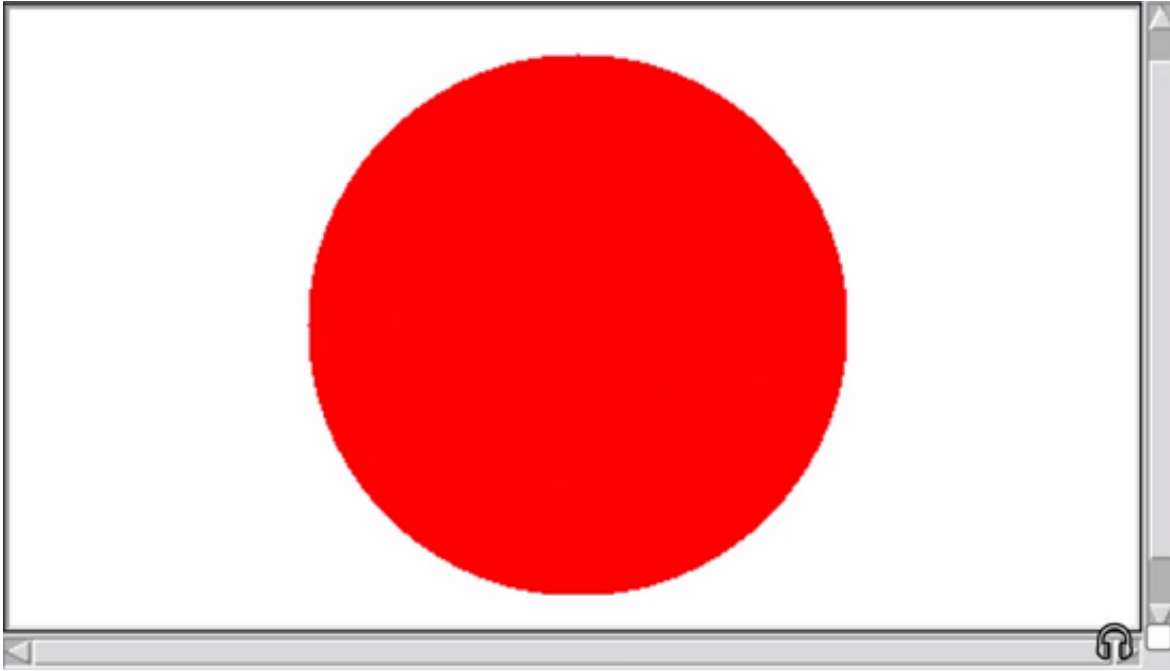
def inflate_balloon():
    global diameter
    diameter = diameter + 10
    draw_balloon()

draw_balloon()

onkey(inflate_balloon, "Up")
listen()
```

It takes two parameters, the first one is a **function** and the second one is the **key** that has to be pressed for the function passed to be called. So, namely, the onkey function **binds** the up arrow key to a callback function which in our case is the inflate_balloon function. Besides, to listen for the input we just defined, we need to call the **listen** function too to keep the program monitoring the inputs received.

Running it, we see that we're able to increase the size of our balloon by pressing the up key as expected:



Note: If the balloon is not inflating, make sure to click on the Output window and then press the Up key.

Popping the Balloon

Lastly, we need to **pop** the balloon when it reaches the `pop_diameter` size as it is only getting continuously bigger now. Let's implement an **if** statement to check if the balloon is ready to pop:

The following code has been updated, and differs from the video:

```
from turtle import *

diameter = 40
pop_diameter = 100

def draw_balloon():
    color("red")
    dot(diameter)

def inflate_balloon():
    global diameter
    diameter = diameter + 10
    draw_balloon()

    if diameter >= pop_diameter:
        clear()
        diameter = 40
        write("POP!")

draw_balloon()

onkey(inflate_balloon, "Up")
listen()
done()
```


After we've inflated our balloon, still inside the `inflate_balloon` function, we check if the diameter of the balloon is **greater** than or **equal** to the `pop_diameter` (100 pixels). When that's the case, we **clear** everything that's on screen, set the **diameter** back to its original **40** pixels, and **write "POP!"** on the screen.

Notice that we're adding the `done()` command so that the Turtle window does not close right after running our program.

Running our code, the balloon keeps inflating and popping over and over again:



Notice that we didn't indicate that we're accessing the global `pop_diameter` variable as we have with the `diameter` variable, and that is because we're **not assigning** it a new value, so Python references the known existing one from the top of our code.

Note: If you make the diameter smaller than 40, you'll be able to see that the "POP!" message is never erased from the screen. To clear it before drawing the new balloon, we need to check that the previous balloon has just popped as follows:

The following code has been updated, and differs from the video:

```
from turtle import *

diameter = 20
pop_diameter = 100
pop = False

# draws the balloon on screen
def draw_balloon ():
    color("red")
    dot(diameter)

# called when we press the Up arrow key
def inflate_balloon ():
    global diameter, pop
    diameter = diameter + 10
    if pop:
        # if balloon just popped we clear the screen before drawing new one
        clear()
        pop = False
```



```
draw_balloon()

# are we ready to pop?
if diameter >= pop_diameter:
    clear()
    diameter = 20
    write("POP!")
    pop = True

draw_balloon()

# call inflate_balloon when we press the Up arrow key
onkey(inflate_balloon, "Up")

listen()
done()
```



The final step of our project is to **evaluate** if we have met the **criteria** we've defined at the start:

- Be able to **detect inputs** – We've successfully done that with the onkey command by binding the key we want to listen for to the function to be called as a callback.
- Require **multiple** inputs to pop the balloon – We've done that with the if statement where we check when the diameter of the balloon is big enough for it to pop, thus having us inflate it multiple times until it does.
- Utilize **variables**, **conditions**, and **functions** – We've used 2 functions along with an if statement and 2 global variables.

And that is our Balloon Popper project complete!

**Instructions:**

Enhance your **Balloon Popper** game by adding **one new simple feature**. Here are some ideas:

- **Change Balloon Color:** Make the balloon change color each time it inflates.
- **Add Deflation:** Make the balloon smaller when a different key is pressed.
- **Adjust Inflation Amount:** Change how much the balloon inflates with each key press.

Choose one idea or come up with your own, and implement it in your game.



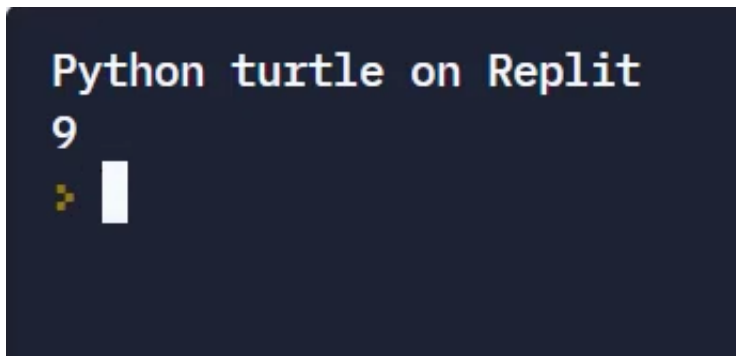
In this lesson, we're going to look at how to use randomness in our algorithms.

So far, every value in our code has been **inputted** by ourselves and so the outcome was **predictable**. However, you may want random sizes or positions for objects, and you don't want to have to place them around manually, one by one. For that, we're going to use Python's **random** library. Let's start by **importing** it:

```
from random import *  
  
random_number = randrange(1, 10)  
print(random_number)
```

The function we're calling is **randrange**, and we need to pass it a **minimum** and a **maximum** value. It'll return to us a random number **between** the range sent as parameter (in this case, a number between 1 and 10).

Running it will give you a **different** result each time:



We can also use it with **Turtle**:

```
from random import *  
from turtle import *  
  
forward(randrange(20, 100))  
right(randrange(0, 360))  
forward(randrange(20, 100))
```

Here, we're moving **forward** a **random** amount of pixels (ranging from 20 to 100), then **rotating** a random angle (anything from 0 to 360°), and **moving** forward again:

And we get a bunch of varied results:



In this lesson, we're taking a look at loops in Python.

Loops allow us to **execute** a block of code **multiple times**.

To show an example of this, let's start by **printing** out a number, **increasing** its value, and **printing** it out once more over and over again:

```
num = 0

num = num + 1
print(num)

num = num + 1
print(num)

num = num + 1
print(num)

num = num + 1
print(num)

num = num + 1
print(num)
```

We can see that it's already becoming quite **repetitive** and we've only done this a couple of times. Instead, we can use a **for loop** to do the exact same thing:

```
num = 0

for x in range(10):
    num = num + 1
    print(num)
```

For this, we write the **for** keyword, the name of a **temporary variable**, then the **in** keyword, and a **range** to define the number of times we want to iterate upon this loop. Lastly, we enter the **colon** symbol and go to a new line to start the indented **block** of code for our for loop. With **only 3 lines of code**, we're now printing out more numbers than we were doing before just by using loops.

Here's the result in the console:



Python turtle on Replit

1

2

3

4

5

6

7

8

9

10



The for loop **runs** our code, **increases x** by **1**, and **repeats** the execution of the code until **x** is out of the specified range when the program moves on with the execution of the rest of the code.



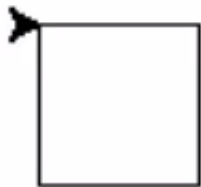
For this **challenge**, let's start with the following code:

```
from turtle import *

def move_and_turn(angle):
    forward(50)
    right(angle)

move_and_turn(90)
move_and_turn(90)
move_and_turn(90)
move_and_turn(90)
```

This code is **creating** a **square**, as we're **moving** the turtle forward and then **rotating** it to its right **90°** each time we call the **move_and_turn** function:



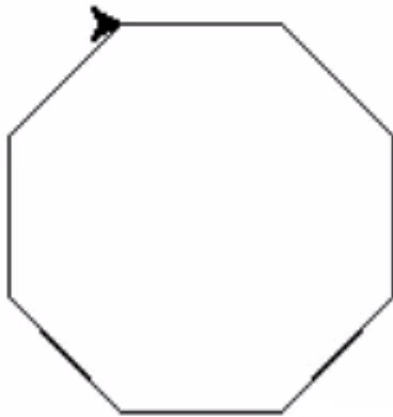
Similarly, we can call our function **8** times passing in **45°** as a parameter instead to make an **octagon** (an eight-sided polygon):

```
from turtle import *

def move_and_turn(angle):
    forward(50)
    right(angle)

move_and_turn(45)
move_and_turn(45)
move_and_turn(45)
move_and_turn(45)
move_and_turn(45)
move_and_turn(45)
move_and_turn(45)
move_and_turn(45)
```

So we have:



As you can see, it's becoming quite a lot of code already and we can simplify all that by using loops. Your task is to **write** the code for the **octagon** above with a **for loop**.

We'll go over the solution in the next lesson!



In this lesson, we'll go over the solution for the loops challenge. As you'll recall, you were tasked with simplifying our code for making an **octagon** using a **for loop**.

Solution

To solve this challenge, we just add a for loop that runs **8 times** our **move_and_turn** function, passing in **45°**:

```
from turtle import *

def move_and_turn(angle):
    forward(50)
    right(angle)

for x in range(8):
    move_and_turn(45)
```

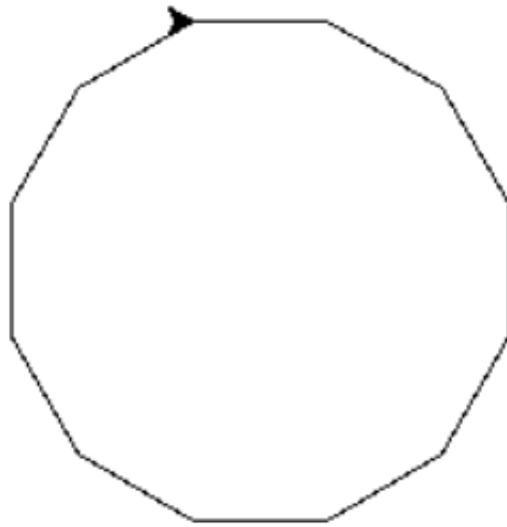
If now we decide to make a **dodecagon** (a twelve-sided polygon), we need only to **change** the values for the **range** to **12** and **angle** to **30**:

```
from turtle import *

def move_and_turn(angle):
    forward(50)
    right(angle)

for x in range(12):
    move_and_turn(30)
```

And here it is:





Loops

Loops are a very important part of programming, since they allow us to run a piece of code as many times as we want. We've looked at the **for loop**, which creates a variable and increases it by 1 each iteration until it reaches the given range.

```
for x in range(10):  
    print(x)
```

A for loop iterates for a set number of times. There's also another type of loop, known as a **while loop**. A while loop will iterate upon the block of code as long as a given condition is true.

```
x = 0  
  
while x < 5:  
    print(x)  
    x += 1
```

Here's how it works.

1. The loop checks to see if the condition ($x < 5$) is true.
2. If so, then the code is executed.
3. Then repeat. Check the condition, run code, etc.
4. When the condition is no longer true, the loop ends.

Now let's say you want to stop a for loop before it ends or stop a while loop before the condition is false. For this, we can use the **break** keyword.

Let's say we have a for loop which loops 100 times.

```
for x in range(100):  
    print(x)
```

When x reaches the number 50, let's end the loop. Here's how we can do that.

```
for x in range(100):  
    if x == 50:  
        break
```

break will automatically end your loop and continue on with the rest of the algorithm.

**Instructions:**

Create a program that uses a loop to draw a simple dashed line using Python Turtle. The line should consist of short segments with gaps in between, resembling a dashed or dotted line.

Tips:

- Use a for loop to repeat the drawing actions.
- Within the loop, use `forward()` to draw a short line segment.
- Use `penup()` and `pendown()` to create gaps between the segments.
- Adjust the length of the line segments and gaps to your preference.



In this lesson, we're starting on our Starry Night Sky project.

The first step in this project is going to be defining the criteria we need to meet.

Definition

Our **goals** with this project are:

1. **Create** a **star system** on a black background.
2. Use **randomness** for the **size** of the stars and their **positions**.
3. Utilize a **for** loop to generate a **large** number of stars.

In the next lesson, we'll design our project and in the sequence jump right into its implementation.

In this lesson, we're doing the design for our Starry Night Sky project.

Diagram

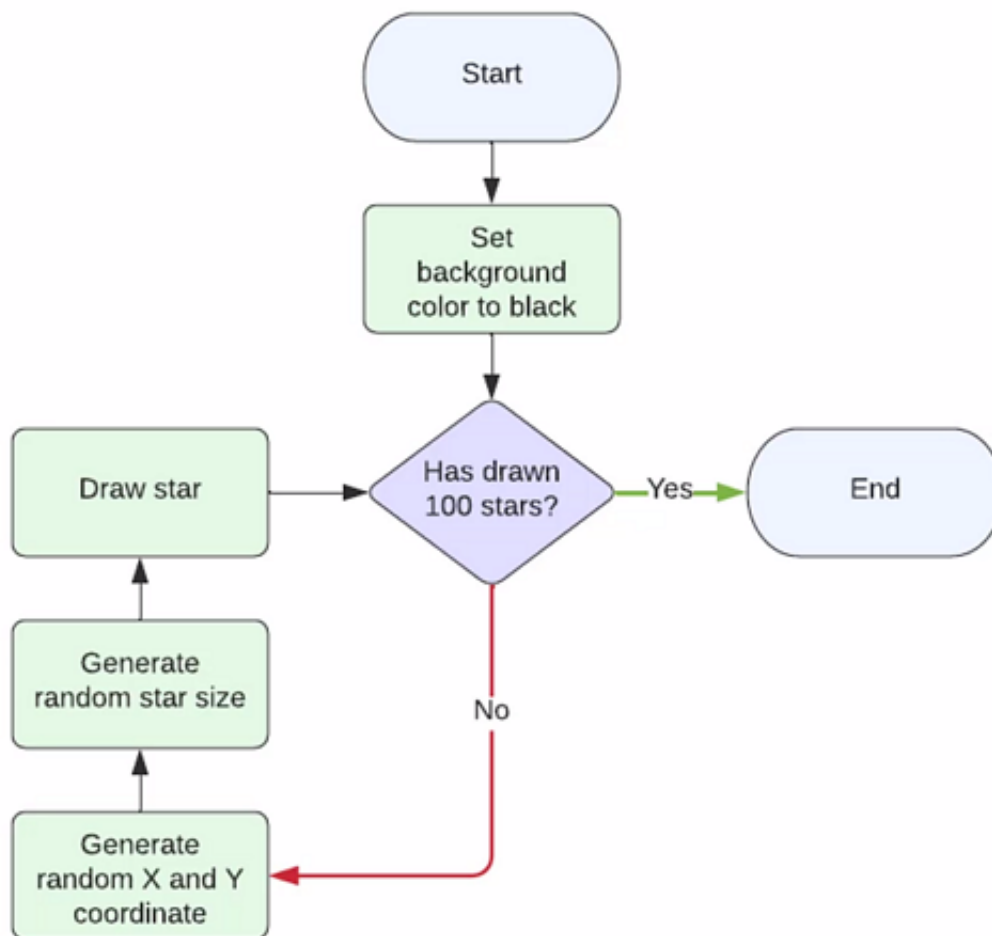
Our final project will look like this:



That is basically a black background with white dots as the stars.

Flowchart

The flowchart is going to show us the general layout of the logic for our algorithm:



We start by setting the **background** color to black to simulate a night sky, then we proceed to check if we have drawn **100 stars** in total. If so, that is our algorithm complete. If **not**, we're going to **generate** a random **x** and **y** coordinate and generate a random star **size** in order to **draw** it onto our black sky, then check the condition again until we've drawn the 100 stars we need.

Pseudocode

Fairly similar to the flowchart, we have the pseudocode for our project as well:

START

Set the background color to BLACK

Repeat 100 times

 Generate random star position

 Generate random star size

 Draw the star

END

We set the background color to black then **repeat** those 3 lines of code 100 times, namely



generating a random **star position** and a random star **size** to **draw** the new star onto the screen until we have completed **100** stars.



In this lesson, we're starting to implement our Starry Night Sky project.

First off, let's import Turtle and **set** our **background** color to **black**:

```
from turtle import *  
  
bgcolor("black")
```

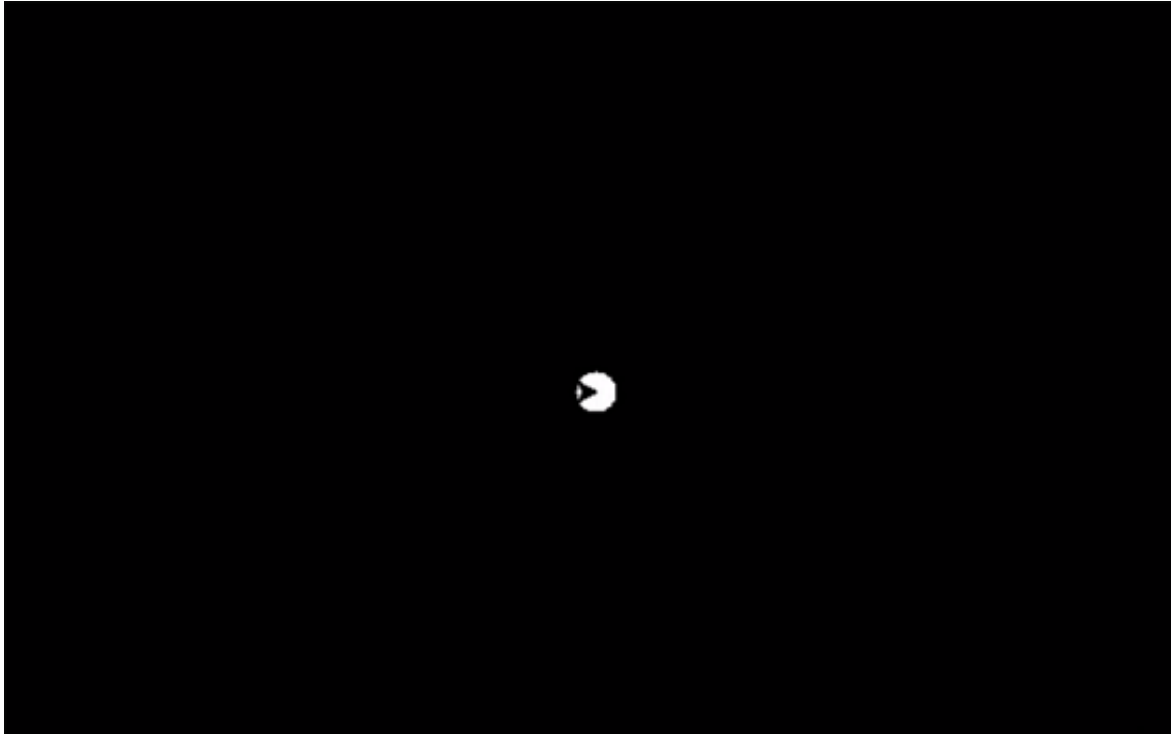
Next, we're going to define a **function** that'll **draw** a **star** onto our black background:

```
from turtle import *  
  
bgcolor("black")  
  
def draw_star(xpos, ypos):  
    penup()  
    goto(xpos, ypos)  
    pendown()  
    dot(20, "white")  
  
draw_star(0, 0)
```

Our `draw_star` function has two parameters: **xpos** and **ypos**. They will determine where on the screen we're going to draw the star, otherwise all stars would be drawn on top of each other at the center of the screen. `xpos` gives us a position on the **x-axis** (e.g. horizontal axis) while `ypos` is for the **y-axis** (vertical axis), this way we can pick any point of the window to place our stars.

The first thing we're doing inside our function is **lifting up** the **pen** so the turtle won't draw a line to its new position while moving. To get there, we're using the **goto** function which receives both `xpos` and `ypos` parameters. Once the turtle has positioned itself in its new location, we **put down** the **pen** again so we can actually **draw** the star by calling the **dot** function. Here, we pass in the **diameter** of the star and the **color** we want it to have.

At last, we **call** our `draw_star` function on position **(0, 0)** to test our code:



Note that the small black shape on top of our star is the turtle itself. To **hide** it, **add** this line after setting the background to black:

```
hideturtle()
```

Let's add a few more stars just to get a feel of how our sky is going to look like by drawing a couple more stars at random positions:

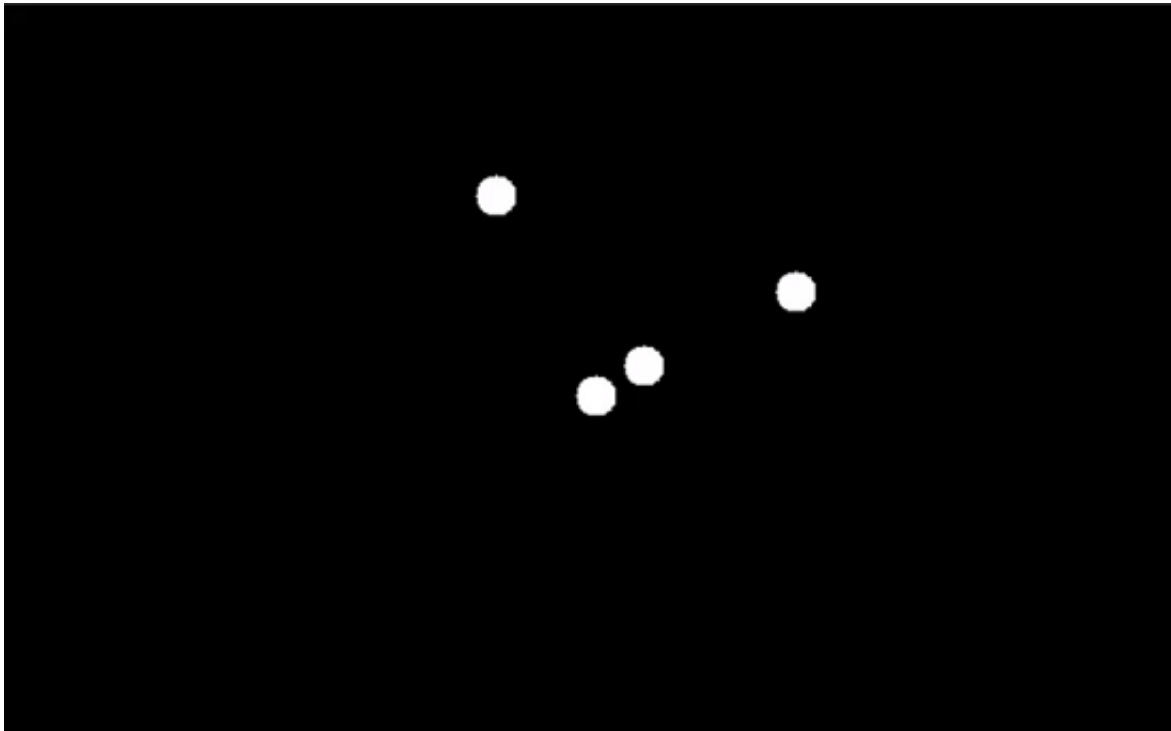
```
from turtle import *

bgcolor("black")
hideturtle()

def draw_star(xpos, ypos):
    penup()
    goto(xpos, ypos)
    pendown()
    dot(20, "white")

draw_star(0, 0)
draw_star(24, 15)
draw_star(-50, 100)
draw_star(100, 52)
```

Here we have our 4 stars:



In this lesson, we'll be implementing randomness into our starry night sky.

Let's **import** Python's **random** library into our code:

```
from random import *
```

Let's then **create** a variable named **size** inside our **draw_star** function and give it a random value between **2** and **7**. Remember that the **randrange** function receives two parameters, a minimum and maximum values, and it returns a random value in between.

When calling the **dot** function, pass in the size variable instead of the fixed value we had before:

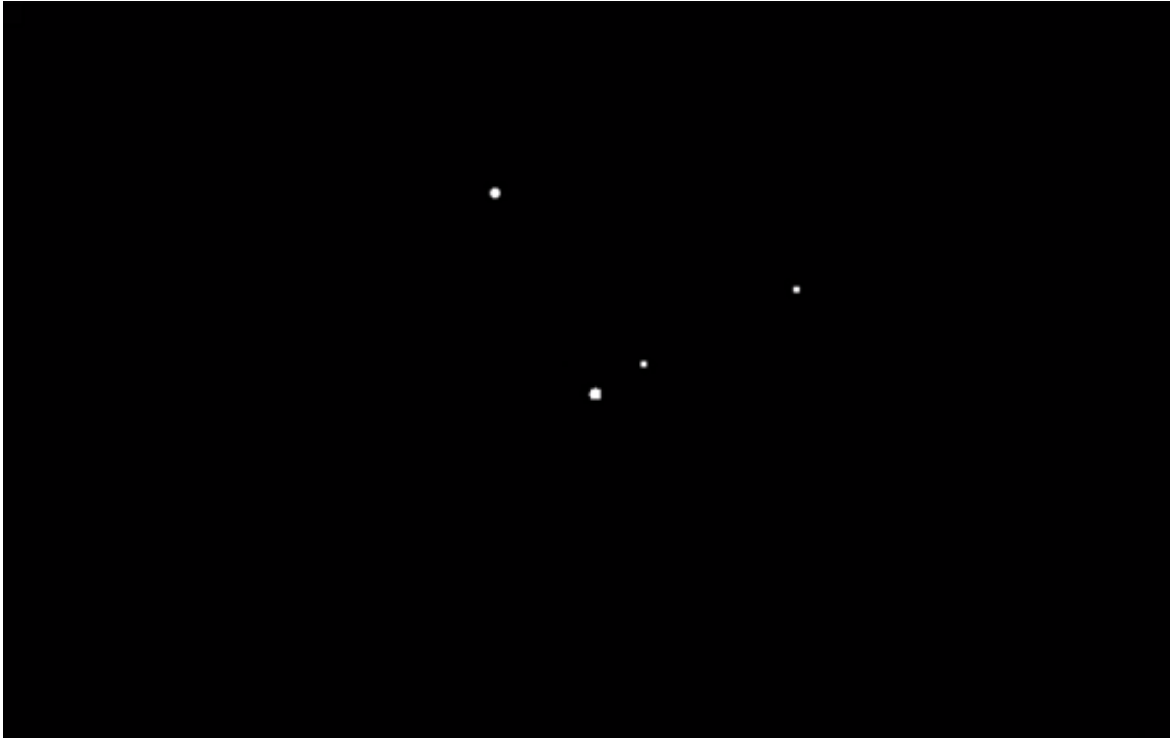
```
from turtle import *
from random import *

bgcolor("black")
hideturtle()

def draw_star(xpos, ypos):
    size = randrange(2, 7)
    penup()
    goto(xpos, ypos)
    pendown()
    dot(size, "white")

draw_star(0, 0)
draw_star(24, 15)
draw_star(-50, 100)
draw_star(100, 52)
```

Now our stars differ in size:



Our next step is to randomly **generate hundreds of stars** in our sky and, for that, we're going to need a **for loop**:

```
from turtle import *
from random import *

bgcolor("black")
hideturtle()

def draw_star(xpos, ypos):
    size = randrange(2, 7)
    penup()
    goto(xpos, ypos)
    pendown()
    dot(size, "white")

for x in range(100):
    print("draw star")
```

Here we have our loop stated, however, we first need to know the **width** and **height** of the **screen** so we can pick a random spot to draw each star. That's what we're going to cover in the next lesson.

In this lesson, we're going to conclude the implementation of our starry sky project.

Let's start by **adding 2 variables** to get the **width** and the **height** of our screen:

```
from turtle import *
from random import *

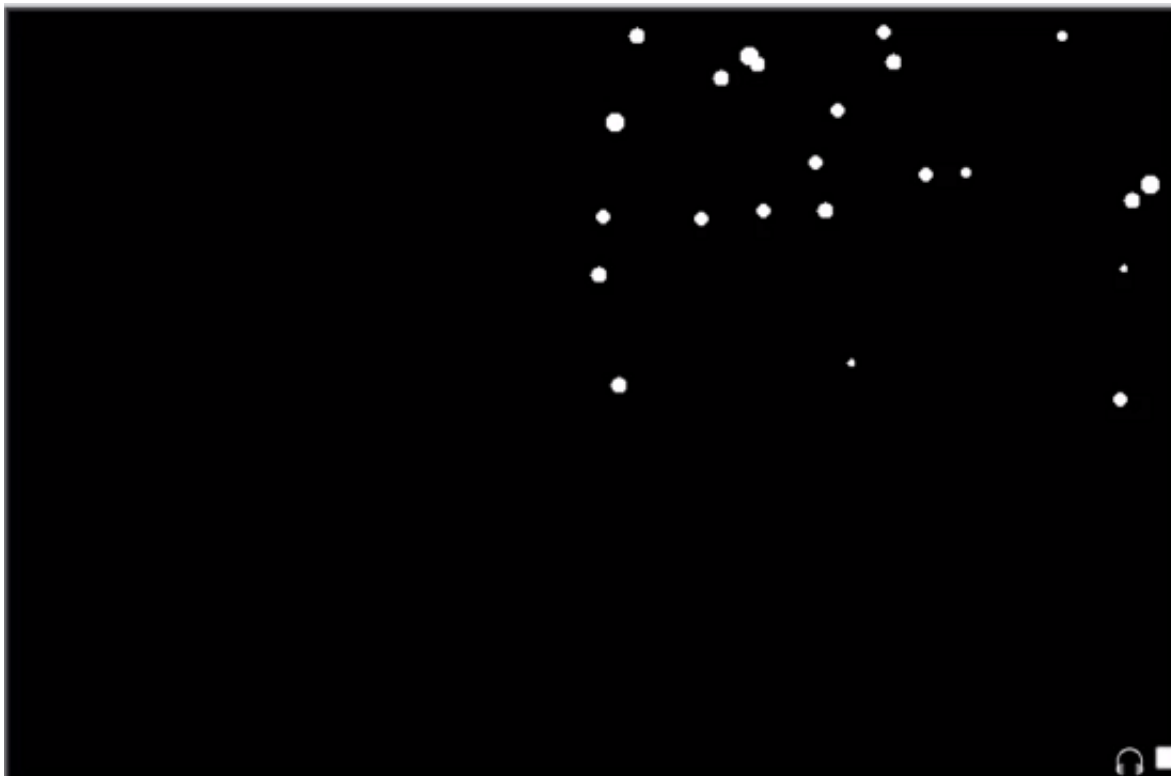
bgcolor("black")
hideturtle()

width = window_width()
height = window_height()

def draw_star(xpos, ypos):
    size = randrange(2, 7)
    penup()
    goto(xpos, ypos)
    pendown()
    dot(size, "white")

for x in range(100):
    xpos = randrange(0, width)
    ypos = randrange(0, height)
    draw_star(xpos, ypos)
```

In our for loop, we can define 2 variables for **xpos** and **ypos** that will range from zero to the corresponding width and height of the screen, then **call** our **draw_star** passing them as parameters:



Note that the stars are only being drawn in the **top-right** corner of the window, though. That is due

to point **(0, 0)** being the **center** of our screen. To properly use the entire screen, we need to **shift** our coordinates half back down and to the left, going over the negative coordinates too as follows:

```
from turtle import *
from random import *

bgcolor("black")
hideturtle()

width = window_width()
height = window_height()

def draw_star(xpos, ypos):
    size = randrange(2, 7)
    penup()
    goto(xpos, ypos)
    pendown()
    dot(size, "white")

for x in range(100):
    xpos = randrange(-width/2, width/2)
    ypos = randrange(-height/2, height/2)
    draw_star(xpos, ypos)
```

Running our code, we see that it's now taking the **whole** size of our **screen** correctly:



NOTE: Although you'll see an error stating that non-integer arguments to `randrange()` have been deprecated since Python 3.10, this issue is solved in the next lesson with the

**use of the *round* function.**

To make it run faster, **add** the **speed** command:

```
from turtle import *
from random import *

speed(0)

bgcolor("black")
hideturtle()

width = window_width()
height = window_height()

def draw_star(xpos, ypos):
    size = randrange(2, 7)
    penup()
    goto(xpos, ypos)
    pendown()
    dot(size, "white")

for x in range(100):
    xpos = randrange(-width/2, width/2)
    ypos = randrange(-height/2, height/2)
    draw_star(xpos, ypos)
```

In this lesson, we're going over a bug that may be occurring with your star system.

When you click on '**Run**', it may sometimes give you an **error** that says that Python's not able to call the `randrange` function. That's because we are **not** giving it an **integer** as a parameter, it may be the case that we're passing in a floating-point number instead if the **width** or the **height** of the screen is **not divisible by 2**.

As the **randrange** function only takes as inputs integers, we need to **round** any possible decimal numbers we get to their **nearest** integers by using the `round` function:

```
from turtle import *
from random import *

speed(0)

bgcolor("black")
hideturtle()

width = window_width()
height = window_height()

def draw_star(xpos, ypos):
    size = randrange(2, 7)
    penup()
    goto(xpos, ypos)
    pendown()
    dot(size, "white")

for x in range(100):
    xpos = randrange(round(-width/2), round(width/2))
    ypos = randrange(round(-height/2), round(height/2))
    draw_star(xpos, ypos)
```

Basically, we just **wrap** each of our parameters for the `randrange` function in **round functions** first, making sure we'll be able to run our code without any errors as all parameters will now be **integers** for sure.



In this lesson, we're **evaluating** whether we've met the **criteria** defined at the start of the project:

- Make a **star system** on a **black background** - We have our black background and we've generated the stars on top of it.
- Use **randomness** for the **star size** and **position** - We've used Python's random library in order to randomly attribute size and position for the stars on our system.
- Have a **for loop** to generate a **large number of stars** - We implemented the for loop to have it place 100 stars in the sky at random positions.

And that is our Starry Night Sky project complete!

Instructions:

Enhance your **Starry Night Sky** by adding **one new simple feature**. Consider these easy ideas:

- **Add a Moon:** Draw a circle in your night sky.
- **Change Background Shade:** Experiment with different background colors to represent dusk or dawn.
- **Add More Stars:** Increase the number of stars to make the sky look fuller.

Select one idea or create your own, and add it to your project.



In this lesson, we're starting on a Turtle mini-game as our final project.

This project is going to be a **top-down game** where we can **move** the **turtle** around to **reach** an end **goal**.

Definition

Let's define the criteria we're going to base our project on:

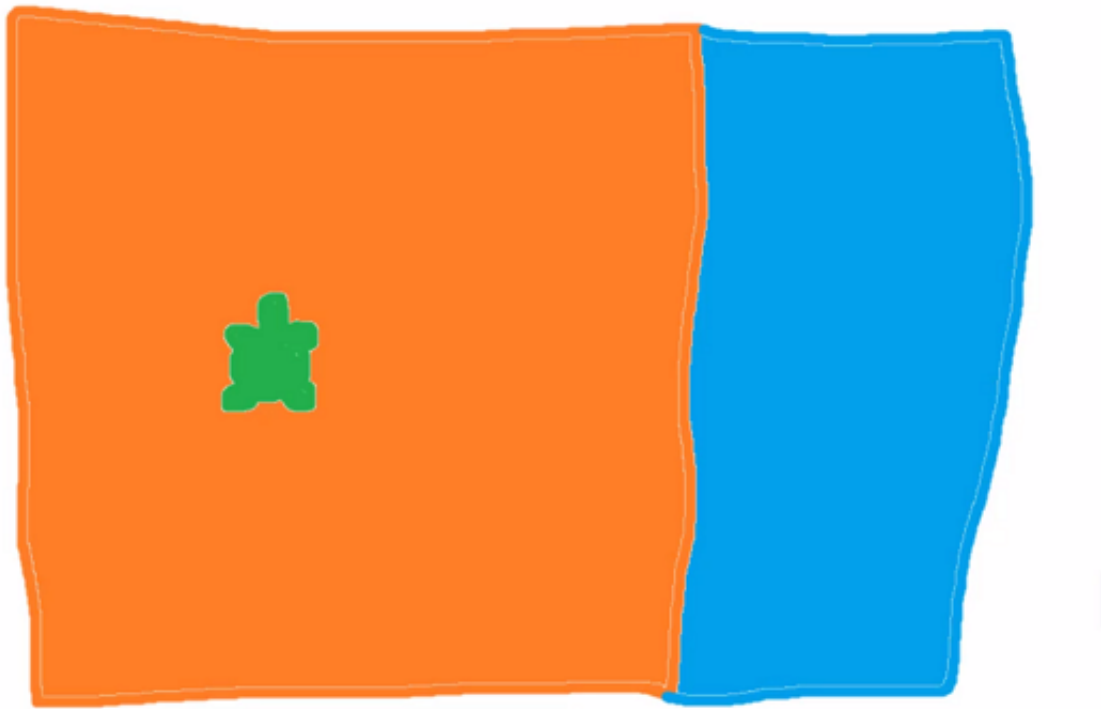
1. Create a **turtle** that can **move** in **4** directions, using the arrow keys (up, down, right, and left).
2. Have an end **goal**. The turtle is born on the beach and has to reach the **ocean**.
3. Have **visual feedback** when the goal is reached so that the player understands that they have won the game.



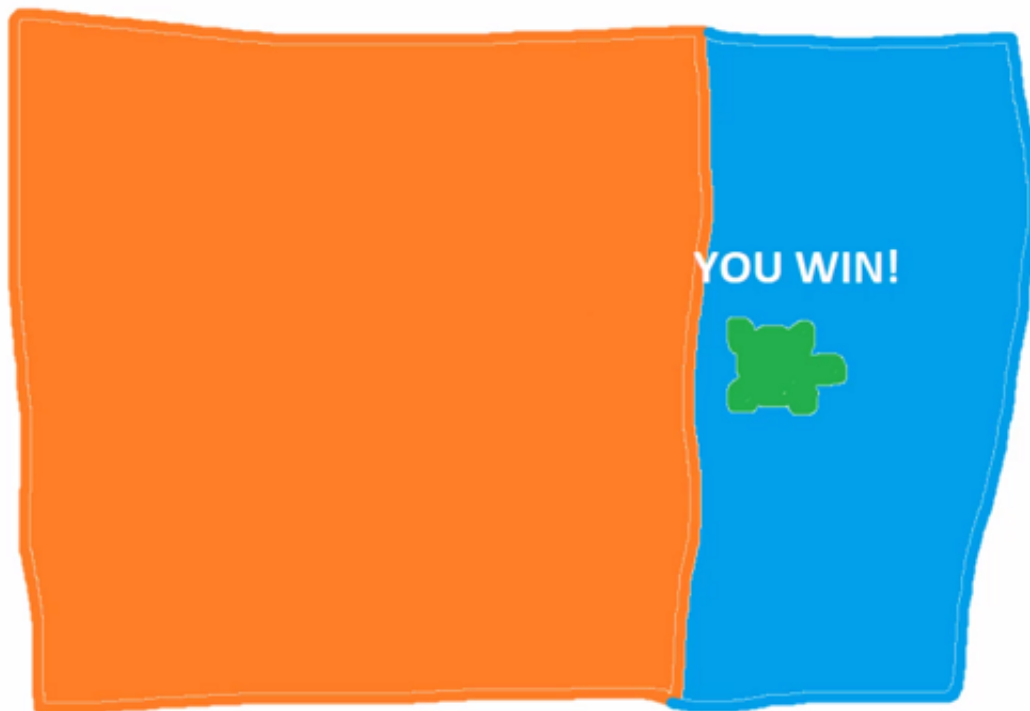
In this lesson, we're designing our Turtle mini-game.

Diagram

Let us start with the diagram, which is going to display the overall look of our project. In the game, we're going to have two main areas: the **beach** (in orange) and the **ocean** (blue). The green turtle is us and we're able to move around as we wish:

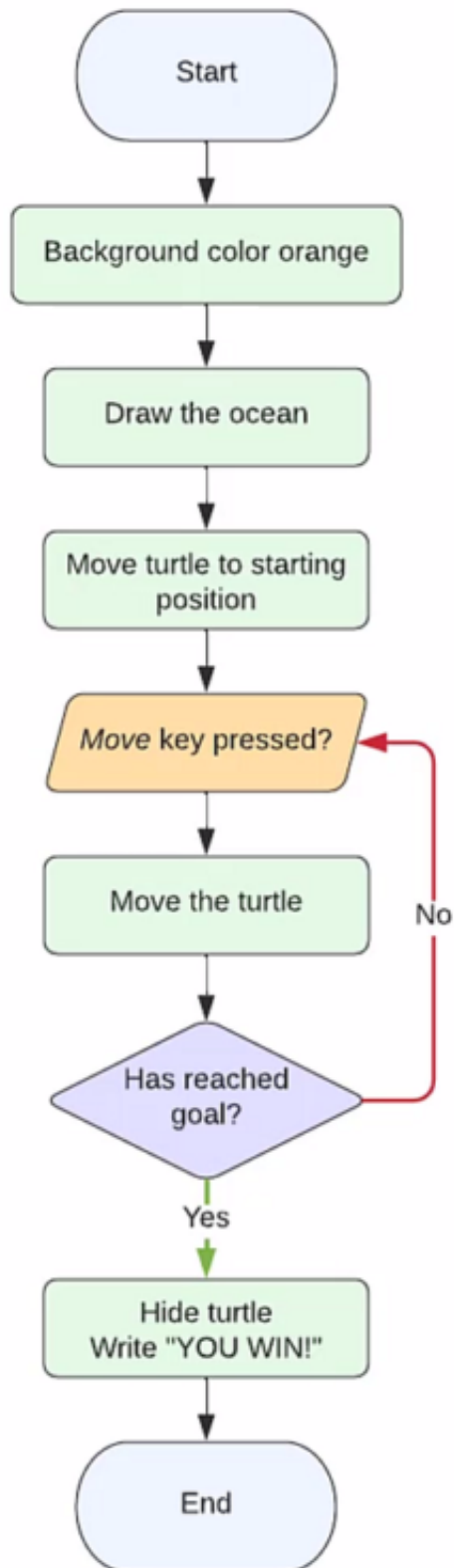


Once we reach the ocean, we're going to see a **"You win!"** message:



Flowchart

Our flowchart is as follows:



We first set the **background** to orange then we create the **ocean** on top of it. Next, we move the turtle to its starting position and we start **listening** for **move** inputs. When one of the 4 **arrow keys** is **pressed** (up, down, left, or right), we move the turtle in that direction and **check** if it has **reached** the **ocean** (i.e., the goal of the game). If so, we hide the turtle and write **"YOU WIN!"** on the screen to show the player they have won the game. In case the goal wasn't reached, we keep listening for

move inputs.

Pseudocode

The pseudocode for our project is very similar to the flowchart above:

START

Set the background color to ORANGE

Draw the ocean

Move the turtle to starting position

Pressed MOVE key?

Move turtle in that direction

Have we reached the goal?

Hide the turtle and disable controls

Write "YOU WIN!" on screen

END

We set the background color to orange, draw the ocean, and move the turtle to the starting position. We proceed to check if any move key has been pressed to move the turtle accordingly, if that's the case, and check whether the player has won the game.

In this lesson, we're starting the implementation of our Turtle mini-game.

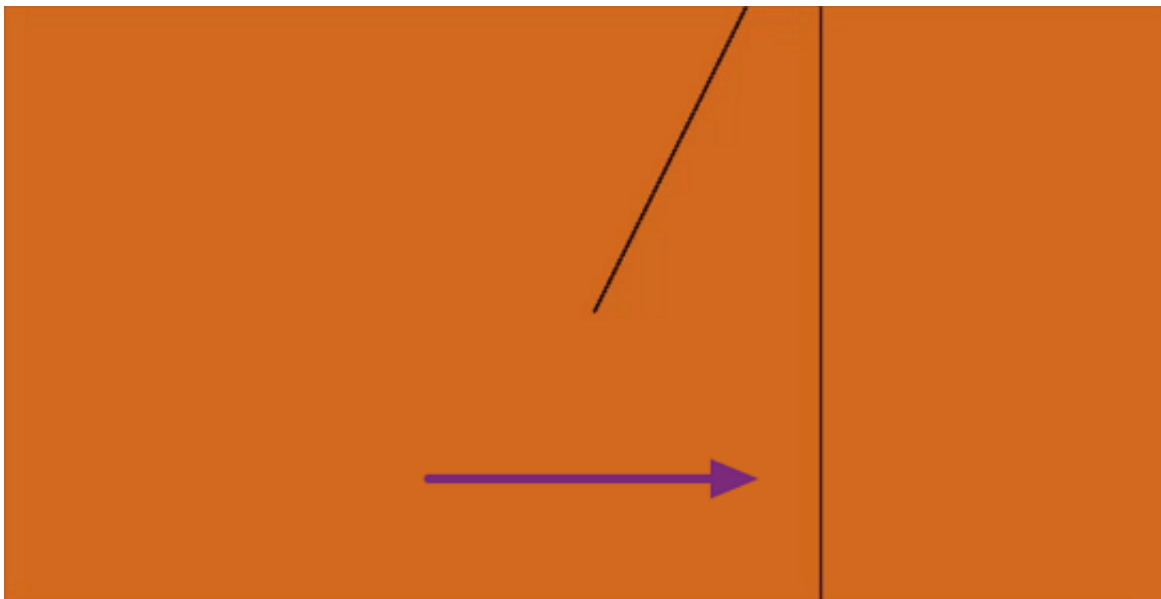
To start, let's import Turtle and declare a variable for the amount we're going to **move** the turtle each time:

```
from turtle import *  
  
move_distance = 20
```

Next, we're setting the **background** color to **orange** by entering the dark orange color in hex below, then we move to the **right-hand** side of the screen to draw the **ocean**:

```
from turtle import *  
  
move_distance = 20  
  
bgcolor("#D2691E")  
  
goto(100, 200)  
  
goto(300, 200)  
goto(300, -200)  
goto(100, -200)  
goto(100, 200)
```

For that, we're going to **draw** a **rectangle** using the **goto** function. The goto function receives two parameters: an **x** position and a **y** position. Our first call to this function moves the turtle to the **top-left** coordinate of the rectangle we're going to draw. Following, we advance to the **three** other points of the rectangle to complete it:



Now, we need to **fill** the rectangle with the **blue** color by setting the color of the turtle to blue. We also need to **encapsulate** the calls for the goto function that we used to draw the whole shape of

the rectangle with the **begin_fill** and **end_fill** commands:

```
from turtle import *

speed(0)
move_distance = 20

bgcolor("#D2691E")

goto(100, 200)

color("blue")

begin_fill()
goto(300, 200)
goto(300, -200)
goto(100, -200)
goto(100, 200)
end_fill()
```

The last thing we need to adjust here is we have to **hide** the **trace** the turtle left behind when moving from the center of the screen to the starting point of our rectangle, so we enclose our first call to the goto function with the **penup** and **pendown** commands:

```
from turtle import *

speed(0)
move_distance = 20

bgcolor("#D2691E")

penup()
goto(100, 200)
pendown()

color("blue")

begin_fill()
goto(300, 200)
goto(300, -200)
goto(100, -200)
goto(100, 200)
end_fill()
```

Running our code, our **beach** and **ocean** are all set up:



In this lesson, we're setting up our turtle in the game.

Continuing with our code, we need to lift up the pen to get the **turtle** back to its **original position**, which is going to be (-200, 0):

```
from turtle import *

speed(0)
move_distance = 20

bgcolor("#D2691E")

penup()
goto(100, 200)
pendown()

color("blue")

begin_fill()
goto(300, 200)
goto(300, -200)
goto(100, -200)
goto(100, 200)
end_fill()

penup()
goto(-200, 0)
shape("turtle")
color("green")
```

Note that we're making the turtle assume the **shape** of an actual **turtle** instead of the usual triangular pointer and, finally, we've also changed its color to **green**:



Moving Our Turtle

To move our turtle around the screen, we'll create 4 functions: `move_up`, `move_down`, `move_right`, and `move_left`.

Let's start by defining the **`move_up`** function first, at the end of our previous code:

```
def move_up():  
    setheading(90)  
    forward(move_distance)  
  
move_up()
```

Here, we're using the **`setheading`** function to **rotate** the turtle to face the **angle** we give it directly. As we don't know its orientation at the moment that we're making it move, we can't use the `right` and `left` commands as we were doing before. Our turtle is at a heading of 0° now, so we pass it **`90`** to make it turn upwards.

Testing our code, we see it's working as expected:



As a challenge, go ahead and try implementing the **`move_down`**, **`move_right`**, and **`move_left`** functions based on the `move_up` one. We'll go over the solution for this in the next lesson!

In this lesson, we'll go over the solution to our challenge and finish setting up our move functions.

Solution

We see it's fairly easy to set up our **move_down**, **move_left**, and **move_right** functions; we just need to **adjust** the **angles** we're passing in as parameters:

```
from turtle import *

speed(0)
move_distance = 20

bgcolor("#D2691E")

penup()
goto(100, 200)
pendown()

color("blue")

begin_fill()
goto(300, 200)
goto(300, -200)
goto(100, -200)
goto(100, 200)
end_fill()

penup()
goto(-200, 0)
shape("turtle")
color("green")

def move_up():
    setheading(90)
    forward(move_distance)

def move_down():
    setheading(270)
    forward(move_distance)

def move_left():
    setheading(180)
    forward(move_distance)

def move_right():
    setheading(0)
    forward(move_distance)
```

Listening to Key Events

Once we have the move functions, we need to **link** them to the actual **keys** using the **onkey** function as seen below:

```
from turtle import *

speed(0)
move_distance = 20

bgcolor("#D2691E")

penup()
goto(100, 200)
pendown()

color("blue")

begin_fill()
goto(300, 200)
goto(300, -200)
goto(100, -200)
goto(100, 200)
end_fill()

penup()
goto(-200, 0)
shape("turtle")
color("green")

def move_up():
    setheading(90)
    forward(move_distance)

def move_down():
    setheading(270)
    forward(move_distance)

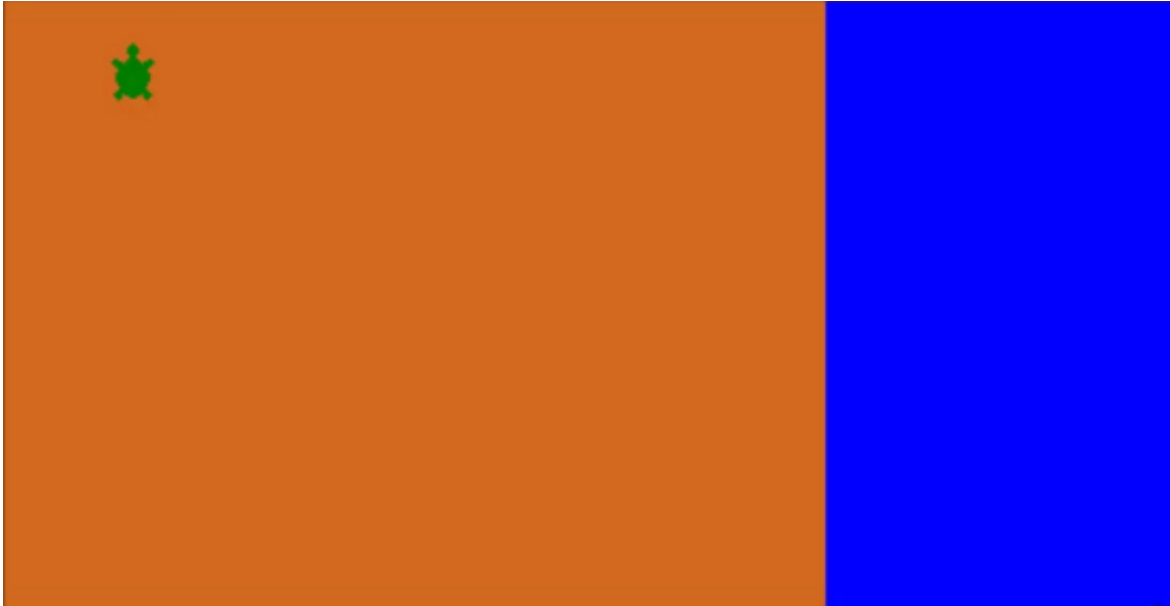
def move_left():
    setheading(180)
    forward(move_distance)

def move_right():
    setheading(0)
    forward(move_distance)

onkey(move_up, "Up")

listen()
```

Then we call the **listen** function in order for our algorithm to keep listening to our stated **events**. This way, we can now **move** our turtle **up** by pressing the up arrow key:





In this lesson, we're going to issue you a challenge.

Last lesson we finished setting up our **movement functions**. We also set up our **onkey** binding for our **up key** and connected that to our **move_up** function.

Your challenge is to complete the other **onkey** bindings for down, left, and right. We'll go over the solution in the next lesson!

In this lesson, we're going to finish implementing our project and go over our challenge solution.

onkey Bindings

First, let's complete the **onkey** bindings for the remaining move functions:

```
from turtle import *

speed(0)
move_distance = 20

bgcolor("#D2691E")

penup()
goto(100, 200)
pendown()

color("blue")

begin_fill()
goto(300, 200)
goto(300, -200)
goto(100, -200)
goto(100, 200)
end_fill()

penup()
goto(-200, 0)
shape("turtle")
color("green")

def move_up():
    setheading(90)
    forward(move_distance)

def move_down():
    setheading(270)
    forward(move_distance)

def move_left():
    setheading(180)
    forward(move_distance)

def move_right():
    setheading(0)
    forward(move_distance)

onkey(move_up, "Up")
onkey(move_down, "Down")
onkey(move_left, "Left")
onkey(move_right, "Right")

listen()
```

Reaching the Goal

To check that we've **reached** the **ocean**, we're going to create a new function called **check_goal**:

```
def check_goal():
    if xcor() > 100:
        hideturtle()
        color("white")
        write("YOU WIN!")
```

As the ocean is starting at 100 on the x-axis, the first thing we need to do is see if the **current x** coordinate of the turtle is **greater** than **100**, using Python's **xcor** function. If that's the case, we make the turtle **invisible**, change the color to white and **write "You win!"** on the screen.

Now, let's check if we're in the ocean after we perform any kind of movement by **calling** our **check_goal** function right after we **move** in **any direction**:

```
from turtle import *

speed(0)
move_distance = 20

bgcolor("#D2691E")

penup()
goto(100, 200)
pendown()

color("blue")

begin_fill()
goto(300, 200)
goto(300, -200)
goto(100, -200)
goto(100, 200)
end_fill()

penup()
goto(-200, 0)
shape("turtle")
color("green")

def move_up():
    setheading(90)
    forward(move_distance)
    check_goal()

def move_down():
    setheading(270)
    forward(move_distance)
    check_goal()

def move_left():
```



```
setheading(180)
forward(move_distance)
check_goal()

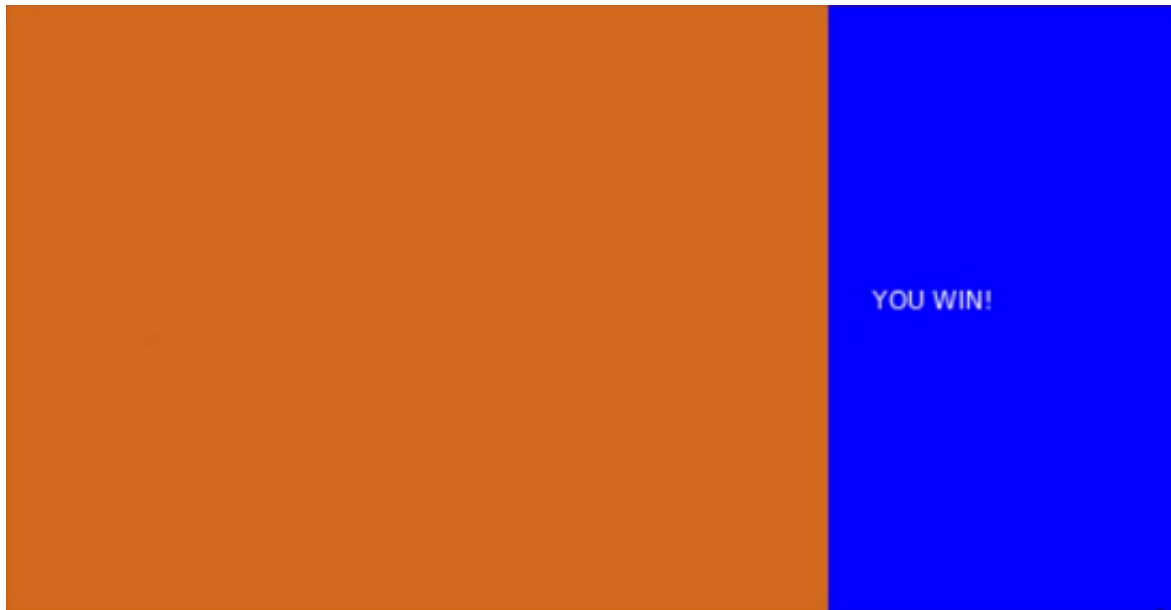
def move_right():
    setheading(0)
    forward(move_distance)
    check_goal()

def check_goal():
    if xcor() > 100:
        hideturtle()
        color("white")
        write("YOU WIN!")

onkey(move_up, "Up")
onkey(move_down, "Down")
onkey(move_left, "Left")
onkey(move_right, "Right")

listen()
```

Run the code to test it:



To avoid continuously writing “You win!” all over the screen if you keep moving the turtle, let’s **disable** the key bindings once we’ve reached our goal as follows:

```
def check_goal():
    if xcor() > 100:
        hideturtle()
        color("white")
        write("YOU WIN!")

        onkey(None, "Up")
```

```
onkey(None, "Down")
onkey(None, "Left")
onkey(None, "Right")
```

We're **resetting** the arrow keys to have no function associated with them anymore by using the **None** keyword. This way, after we win the game, we can no longer move the turtle.

Here's our final code:

```
from turtle import *

speed(0)
move_distance = 20

bgcolor("#D2691E")

penup()
goto(100, 200)
pendown()

color("blue")

begin_fill()
goto(300, 200)
goto(300, -200)
goto(100, -200)
goto(100, 200)
end_fill()

penup()
goto(-200, 0)
shape("turtle")
color("green")

def move_up():
    setheading(90)
    forward(move_distance)
    check_goal()

def move_down():
    setheading(270)
    forward(move_distance)
    check_goal()

def move_left():
    setheading(180)
    forward(move_distance)
    check_goal()

def move_right():
    setheading(0)
    forward(move_distance)
    check_goal()
```



```
def check_goal():
    if xcor() > 100:
        hideturtle()
        color("white")
        write("YOU WIN!")

        onkey(None, "Up")
        onkey(None, "Down")
        onkey(None, "Left")
        onkey(None, "Right")

onkey(move_up, "Up")
onkey(move_down, "Down")
onkey(move_left, "Left")
onkey(move_right, "Right")

listen()
```

To **restart** the game, just **rerun** the program.



Here we're going to **evaluate** what we originally defined in our **criteria** to check if we've reached what we've established:

- Create a **turtle** that can **move** in **4 directions** - We set up four different functions to move the turtle according to the arrow key that was pressed.
- Have an **end goal** - We drew the ocean and continuously check to see if the player had reached it by moving the turtle around the beach.
- Have **visual feedback** when we reach the goal - We disable the turtle so it's no longer visible when it reaches the ocean and we place a message on the screen letting the player know that they won the game.

And that is our Turtle mini-game project complete!

**Instructions:**

Improve your **Turtle Mini-Game** by adding **one simple feature**. Here are some easy ideas:

- **Add Boundaries:** Prevent the turtle from moving off the screen edges.
- **Change Turtle Color:** Make the turtle change color when it reaches the ocean.
- **Display Start Message:** Show a “Help the turtle reach the ocean!” message at the beginning.

Choose one idea or think of your own, and incorporate it into your game.

In this lesson, you can find the full source code for the project used within the course. Feel free to use this lesson as needed – whether to help you debug your code, use it as a reference later, or expand the project to practice your skills!

You can also download all project files from the **Course Files** tab via the project files or downloadable PDF summary.

Variables.py

Found in the Project root folder

The code assigns values to various variables: an integer to 'age', a float to 'temperature', a string to 'country', and a boolean to 'is_day'. After printing the original value of 'country', it then changes the value of 'country' to "Ireland" and prints the new value.

```
age = 20
temperature = 25.39
country = "Australia"
is_day = True

print(country)

country = "Ireland"

print(country)
```

VariablesChallenge.py

Found in the Project root folder

This code defines four variables: 'country_name' for the name of a country, 'population' for its population count, 'landlocked' for its geographical condition (whether it is surrounded by land), and 'border_size' indicating the length of the country's borders. The values provided are specific to Australia.

```
country_name = "Australia"
population = 25000000
landlocked = False
border_size = 60000.256
```

Operators.py

Found in the Project root folder

This code represents a health system in a game, where a player's health decreases by damage taken, increases when health items are obtained, and may be affected by multiplication and division factors. The effects on the health value are displayed using print statements.

```
health = 100
damage = 90
```

```
# subtraction
health = health - damage
print(health)

# addition
health = health + 50
print(health)

# multiplication
health = health * 1.5

# division
health = health / 2
```

Conditions.py

Found in the Project root folder

This code compares two variables, 'a' and 'b'. Initially, it checks if they are equal and updates the value of 'a', then it checks if 'a' is greater than or less than 'b', printing a corresponding message each time the condition is met.

```
a = 5
b = 5

if a == b:
    print("a is equal to b")

if a > b:
    print("a is greater than b")

if a < b:
    print("a is less than b")
```

Loops.py

Found in the Project root folder

This code initializes a variable `num` to 0 and then increments it by 1 for each number in a range of 100, from 0 to 99. The current value of `num` is printed in every step of the loop.

```
num = 0

for x in range(100):
    num = num + 1
    print(num)
```

LoopsChallenge.py

Found in the Project root folder

This Python script utilizes the 'turtle' graphics module to draw an octagon. It defines a function, 'move_and_turn', that moves the turtle forward by a fixed distance and then makes it turn through a specified angle. The function is called 8 times in a loop to form the eight sides of an octagon.

```
from turtle import *

def move_and_turn (angle):
    forward(50)
    right(angle)

# draw an octagon
for x in range(8):
    move_and_turn(45)
```

Functions.py

Found in the Project root folder

This code does two main tasks. First, it defines a function 'say_hello' that prints a greeting message to named individuals. It then calls this function for different names. Second, via the turtle module, it defines a function 'move_and_turn' to move the turtle a certain distance and then turn it by a specific angle, while also printing a message to the console. It calls this function with different arguments.

```
from turtle import *

def say_hello (name):
    print("How are you " + name)

say_hello("Bob")
say_hello("Steve")
say_hello("Mary")
say_hello("Rob")

def move_and_turn (distance, angle):
    forward(distance)
    right(angle)

move_and_turn(100, 45)
move_and_turn(50, 90)
```

Randomness.py

Found in the Project root folder

The code uses the 'turtle' graphics module and the 'random' module in Python to move the turtle. It makes the turtle move forward a random distance (between 20 and 100 units), turn to a random angle (between 0 and 360 degrees), and then move forward again by a random distance (between 20 and 100 units).



```
from random import *
from turtle import *

forward(randrange(20, 100))
right(randrange(0, 360))
forward(randrange(20, 100))
```

BalloonPopperProject.py

Found in Project/Mini Projects

This script uses Python's turtle module to visually represent a "balloon" inflating each time the Up arrow key is pressed. It increases the size (diameter) of the balloon and re-draws it. If the diameter exceeds a preset limit, the balloon "pops" (disappears), a pop message is displayed, and the balloon is reset to its initial size.

```
from turtle import *

diameter = 40
pop_diameter = 100

# draws the balloon on screen
def draw_balloon ():
    color("red")
    dot(diameter)

# called when we press the Up arrow key
def inflate_balloon ():
    global diameter
    diameter = diameter + 10
    draw_balloon()

# are we ready to pop?
if diameter >= pop_diameter:
    clear()
    diameter = 40
    write("POP!")

draw_balloon()

# call inflate_balloon when we press the Up arrow key
onkey(inflate_balloon, "Up")

listen()
```

StarryNightSkyProject.py

Found in Project/Mini Projects

This code uses the 'turtle' and 'random' modules to create a starry sky. It generates 100 randomly sized and positioned stars in a black window. For each star, it calculates a random X and Y position



within the boundaries of the window and calls a function `draw_star` to draw a star at that position.

```
from turtle import *
from random import *

speed(0)
bgcolor("black")
hideturtle()

# get the width and height of the window
width = window_width()
height = window_height()

# draws a star at a given position
def draw_star (xpos, ypos):
    # set a random size for the star
    size = randrange(4, 10)

    # goto the desired position
    penup()
    goto(xpos, ypos)
    pendown()

    # draw the star
    dot(size, "white")

for x in range(100):
    # create a random X & Y position
    xpos = randrange(round(-width / 2), round(width / 2))
    ypos = randrange(round(-height / 2), round(height / 2))

    # draw the star at that position
    draw_star(xpos, ypos)
```

TurtleMiniGameProject.py

Found in Project/Mini Projects

This code uses the turtle module to create a simple “Cross the Sea” game. A beach and ocean are drawn, and the player uses the arrow keys to control a green turtle character, trying to get it from the beach to the sea. When the turtle reaches the sea (X coordinate greater than 100), a “YOU WIN!” message is displayed and the turtle’s movements are disabled.

```
from turtle import *

speed(0)

move_distance = 20

# create beach
bgcolor("#D2691E")

# draw ocean
```

```
penup()
goto(100, 200)
pendown()

color("blue")

begin_fill()
goto(300, 200)
goto(300, -200)
goto(100, -200)
goto(100, 200)
end_fill()

# set turtle starting position
penup()
goto(-200, 0)
shape("turtle")
color("green")

def move_up ():
    setheading(90)
    forward(move_distance)
    check_goal()

def move_down ():
    setheading(270)
    forward(move_distance)
    check_goal()

def move_left ():
    setheading(180)
    forward(move_distance)
    check_goal()

def move_right ():
    setheading(0)
    forward(move_distance)
    check_goal()

# called when we move the turtle
def check_goal ():
    if xcor() > 100:
        hideturtle()
        color("white")
        write("YOU WIN!")

        onkey(None, "Up")
        onkey(None, "Down")
        onkey(None, "Left")
        onkey(None, "Right")

# key press events
onkey(move_up, "Up")
onkey(move_down, "Down")
onkey(move_left, "Left")
```



```
onkey(move_right, "Right")
```

```
# listen for key presses
```

```
listen()
```