



In this lesson, we're going over the definition of programming.

Programming is the process of **writing code** to **instruct** a computer to **do** certain **tasks**. We, as humans, communicate with computers through coding which basically is **a set of instructions** that we give the **computer** to **read** and **execute**.

## What are Algorithms?

An algorithm is a **list of steps/rules** to follow in order to **achieve** a **result**, being the basis of computer programs. Coding algorithms allows us to **solve problems** and **process data**. Simple examples of algorithms we deal with daily are:

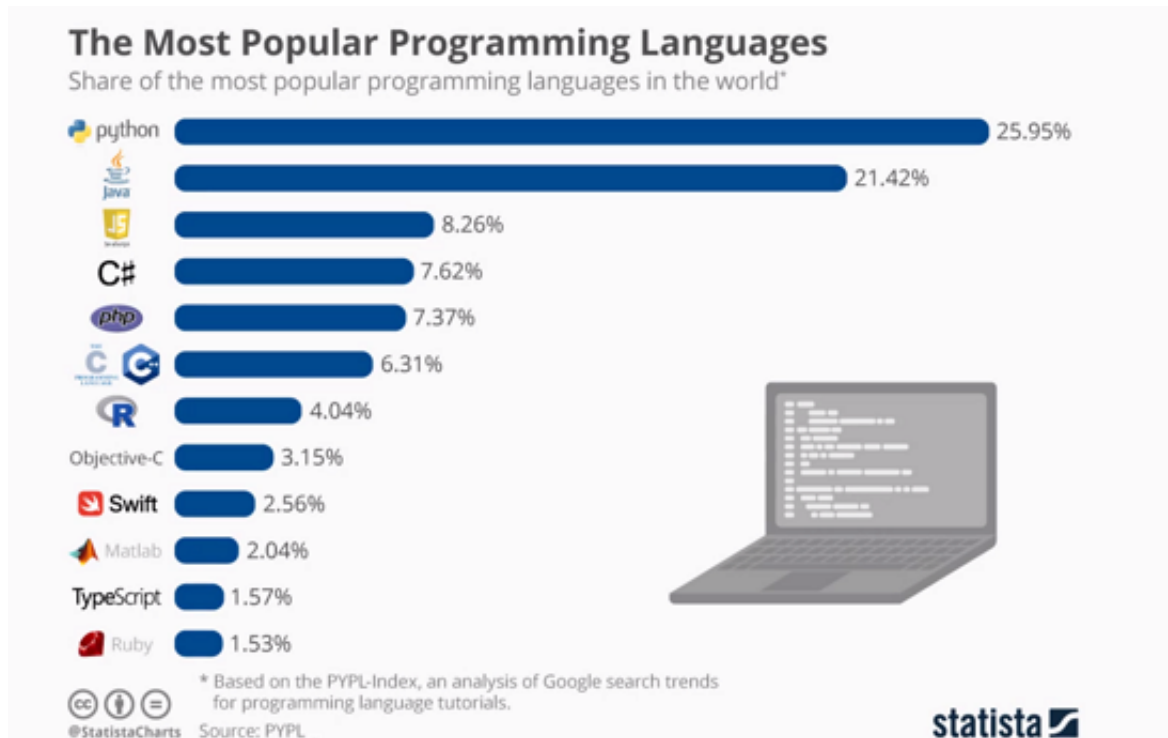
- Recipes
- Bus schedules
- Traffic light systems

Every day we use algorithms when we're saving documents to our devices or loading webpages without even noticing.

In this lesson, we'll go over why we're using Python.

To be actually able to **write code** we need a **programming language**. Just like we have several languages we can speak, such as English, Spanish, or Japanese, we can also instruct computers using a variety of different programming languages.

**Python** is one of the **most popular** languages used nowadays as we can see here:



Due to its **versatility**, Python is used in a wide set of industries, among them:

- Web development
- Video games
- Medical sciences
- Finance
- Space exploration
- Data processing
- Machine learning

Besides, having a **clean** and simple **appearance** as well as being **easy** to **understand** makes it perfect for those just starting out with programming.

## What is a Development Environment?

Regardless of the programming language, **code** is usually written using a type of software called a **code editor**. There are numerous code editors out there, both free and paid. The most popular code editor is called **Visual Studio Code** and is free.

While most code editors are downloadable programs, some of them, such as **Replit**, are hosted in the cloud and can only be used using a web browser.

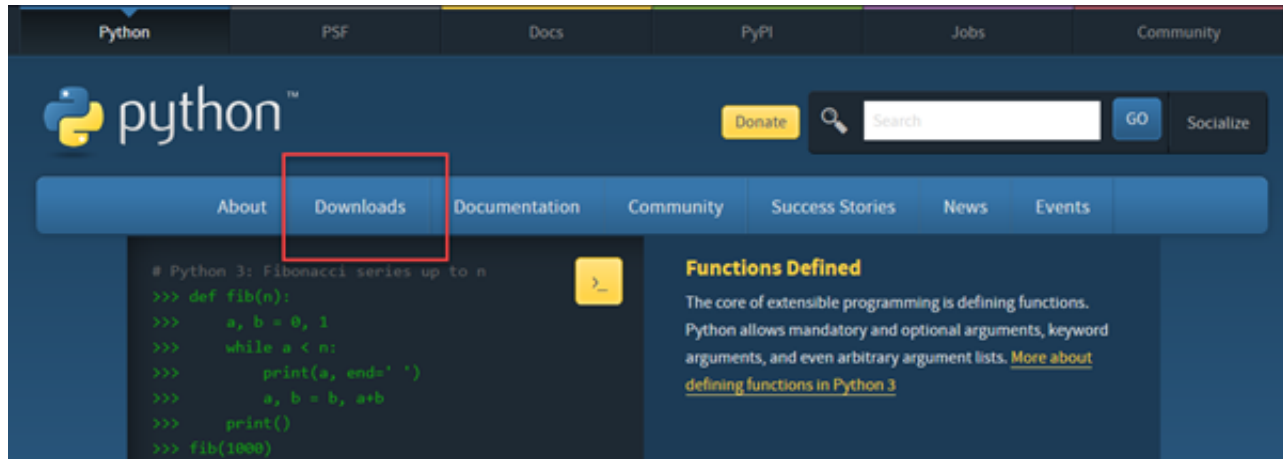
Common features of code editors include:

- Syntax coloring to improve code readability
- Auto-completion capabilities
- File management
- Ability to run code
- Support for different programming languages



In addition, in order to run or execute Python code, you will need access to a **Python runtime**. That being, a program that can take your Python code (which is just text), translate it into executable commands, and have your machine run it.

Python can be downloaded for all operating systems (Windows, Mac, Linux) on the [official Python website](https://www.python.org/).



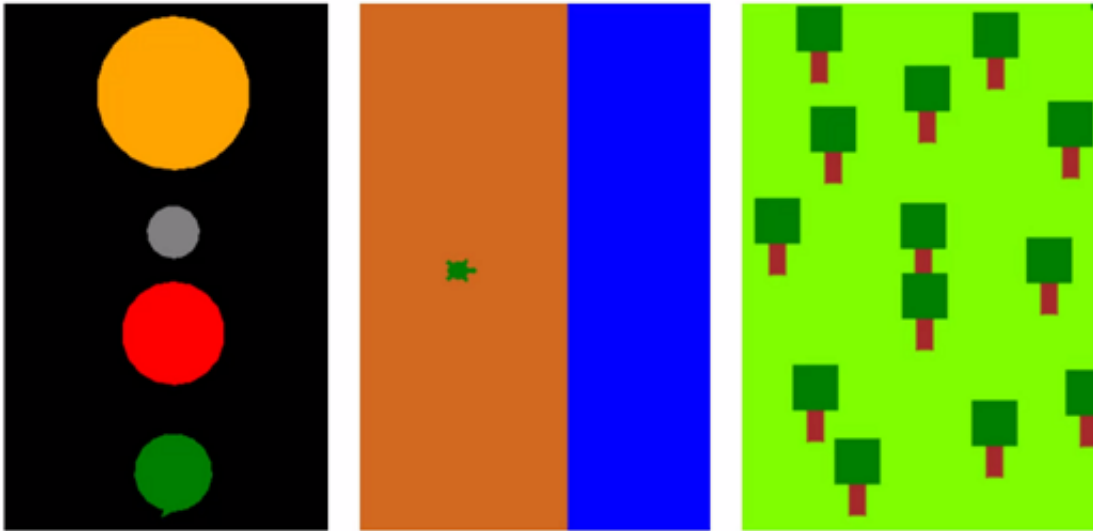
## What You'll Need for This Course

You can follow along any of our Python courses with **any code editor of your choosing**.

This course in particular was recorded using [Replit](https://replit.com), as that requires no installation for the user. In saying that, the lessons have been created so that learners can complete all of the activities using any Python development environment. That means you can complete this course using the code editor of your choice ([Visual Studio Code](https://code.visualstudio.com), Anaconda, etc), as long as you have the Python runtime installed on your computer.

In this lesson, we're going to have a quick look at what Python Turtle is.

**Turtle** is a **Python library** with which we can **create** simple **algorithms**, **digital art**, and **interactive programs**. Through a series of commands, we can **control** a **point** on the screen in order to **draw** and **fill** in **shapes**:



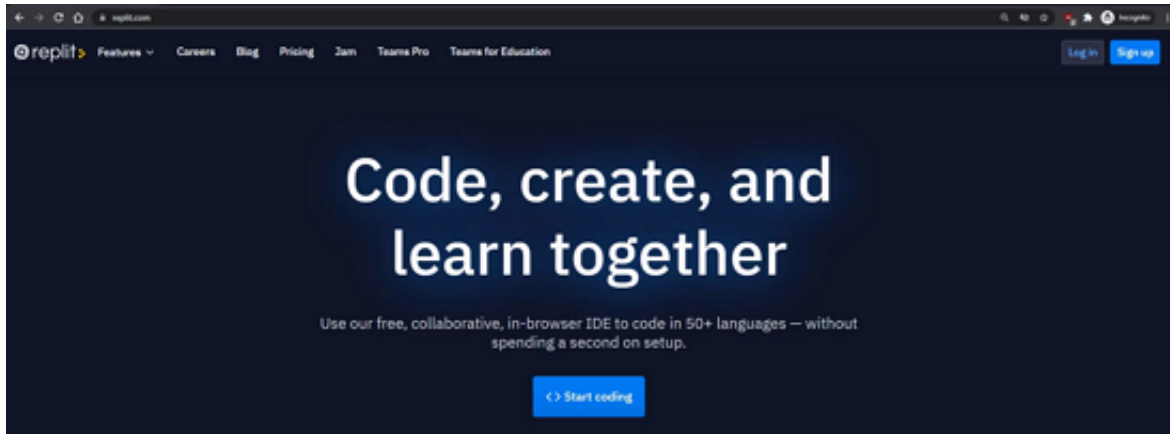
We can compare it to the tip of a pen against a paper, where we command its direction and what to draw.

Learning to program with Python Turtle is a great way of getting the **fundamentals** of how algorithms work while **visualizing** what the code is doing at the same time. So, for instance, if we write a command to draw a circle we'll see the circle being drawn instantly on the screen. And that's why we're using this library throughout the course as it makes coding so **practical** and **simple**.

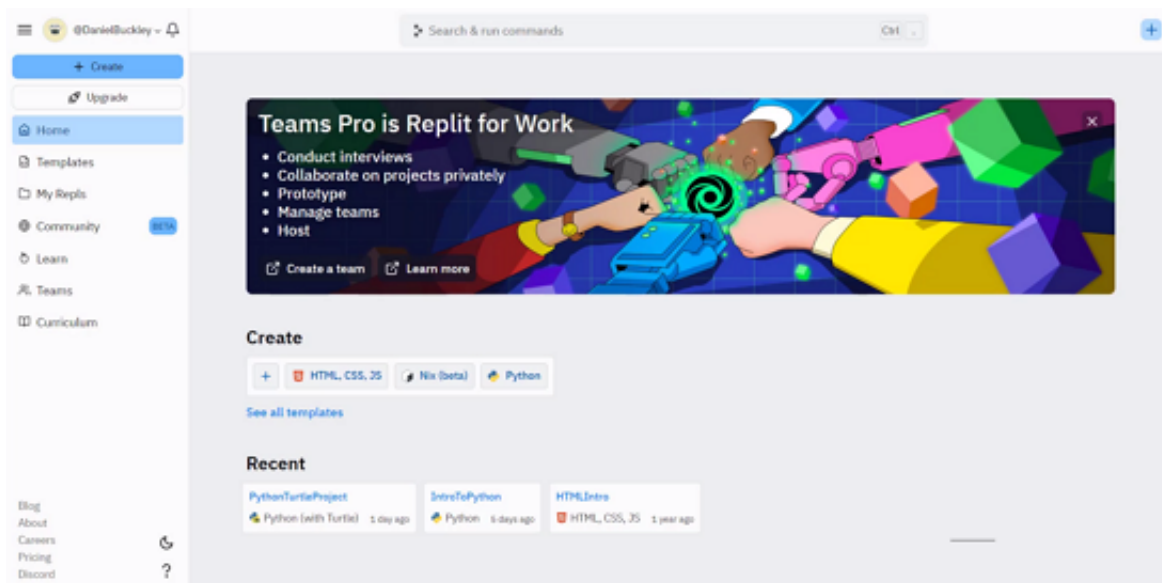
In this lesson, we'll see how to set up Replit.

**Replit** is a **code editor** used **directly** via **browser**. It allows you to write and run code without the need to download any external software to your device.

To get started, go to the [Replit website](https://replit.com):



Go to the **top-right** corner and **click** on '**Sign up**'. This will take you through the needed steps to create a Replit account. Once you fill out the requested form and **log in**, you'll be taken to the **dashboard**:



Here, we can create and manage our different projects. To create a **new project** go to the '**Create**' **button** at the **top** of the **side menu**. Once it opens up, we can **choose** the **template** (i.e., the **programming language**) we want to use. In our case, select '**Python with Turtle**'.

Next, give your project a **name** then **click** on '**Create Repl**':

Create a repl

Import from GitHub

Template: Python (with Turtle) ▼

Title: PythonTurtle

Privacy: Repl is public (toggle off)

Anyone can view and fork this repl

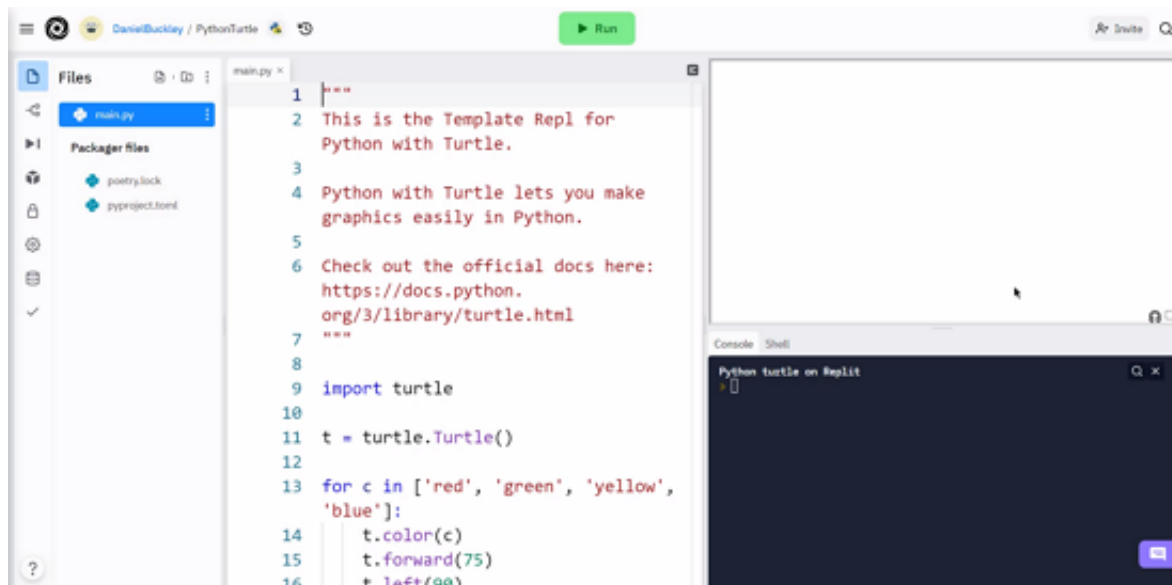
Upgrade to make private

Python (with Turtle) ✓

A simple version of Python that supports Turtle. (Nix Beta)

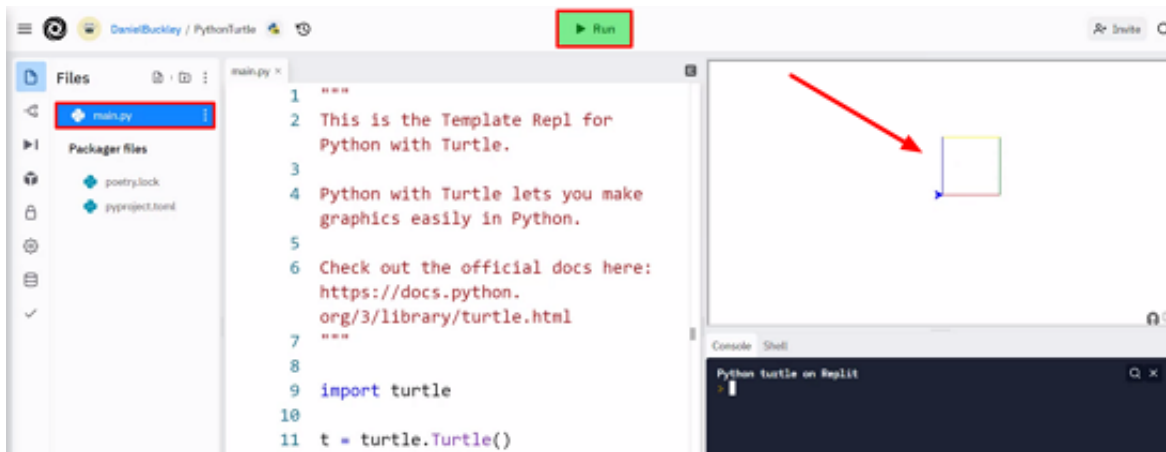
+ Create Repl

Now we're inside our project:



- On the **left-hand** side, there's the **Files panel** listing all the files in our project. Note that when writing code for Python and many other programming languages, the file containing the code will have a **specific file extension** relating to that specific programming language. The extension for the **Python** language is **.py**. Thus, all files ending with **'py'** are Python files containing Python code.
- Over the **center** of the screen, where the **'main.py'** file is now open, is where we'll **write** our code. We can see that some **keywords** are **highlighted** with different **colors** so that the code is easier to read.
- To the **top-right**, the **"Turtle window"** will show the **graphic result** of our code, and below it, we have the **console** window to the **bottom right**. Any errors in your code or in the execution of the program will be displayed here.

Finally, **click** on **'Run'** at the very **center** of the **top** bar to run the code of the **selected** file:







If you're having trouble with your code because the output window closes as soon as your program stops running, try adding the `done()` command at the end of your code to avoid that from happening.

In case you're having issues with the output window size or scrolling, among similar problems, try moving to the code editor of your choice as presented in [this lesson](#).

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 get started on learning **Python Turtle**.

First of all, **delete** the **sample** code that's currently in the **main.py** file so we can start from scratch. Next, let's **import Turtle** as it is a **library** implemented into the Python language and by adding it to our project we can use its **functionalities** inside our program. For that, type:

```
from turtle import *
```

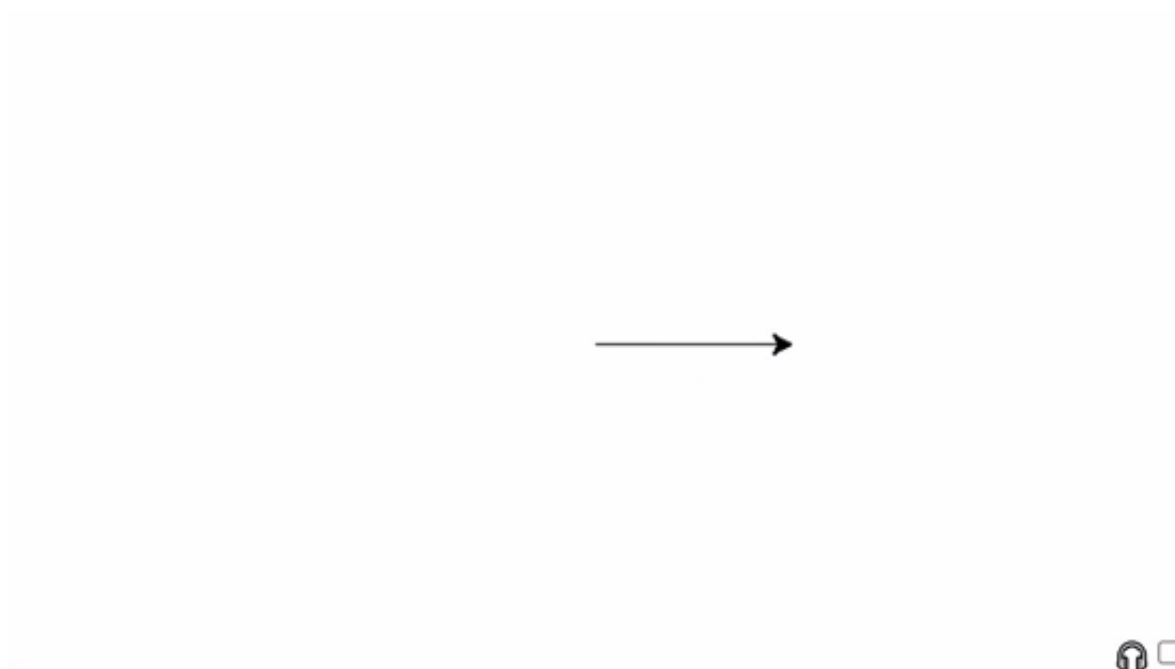
This line of code means that we're importing **everything** (\*) from the Turtle library. Make sure the imports are the **first** thing you have coded in your file as algorithms execute each line of code sequentially, one at a time, from top to bottom thus this way you're making sure you have all you'll need further on.

Now that we've imported Turtle, we have to actually utilize it. Turtle has a large list of **commands** for us to draw different things on the screen, the most common one among them being the **forward** function. To call it, do:

```
forward(100)
```

To move forward just write '**forward**' and pass the **number** of **pixels** you want to move in between parentheses (in our case, **100**).

Click the '**Run**' button to see that Turtle drew a straight **line** starting from the middle of the screen and moving forward to the **right** 100 pixels.



Notice that Turtle works as a **top-down view** of a turtle on the ground facing to the right of our screen, which is its **default** 'forward' direction. If we'd like to move it in another direction, we need



to command it to do so as we'll see in the next lesson.

**NOTE:** To prevent the window from closing after running your code use the ***done()*** command as follows.

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

```
from turtle import *  
  
forward(100)  
done()
```

You could also use *mainloop()* which does the same as *done()*, as the two commands are identical.

Also, if you're having trouble importing turtle as shown in the lesson, try the following alternative version:

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

```
import turtle  
  
t = turtle.Turtle()  
t.forward(100)  
  
turtle.done()
```

In this case, remember to always add *t.* in front of your turtle movement commands and *turtle.* in front of the commands you use from the turtle library.



In this lesson, we'll see how to make our turtle **turn in a different direction**.

So far we've imported the Turtle library and we have moved our turtle forward a hundred pixels to the right with the following code:

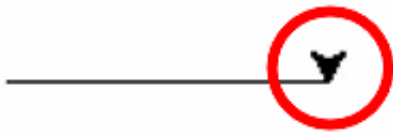
```
from turtle import *  
  
forward(100)
```

Let's add more steps to our algorithm above by **adding a new line** (after the forward command) to **turn** the turtle to the **right**:

```
right(90)
```

Here, rather than entering the number of pixels we want the turtle to move, we have to pass the **angle** that we wish to make the turn to the right.

Click on '**Run**' to see that the **arrow point** is now facing down at the end of the line, after **rotating 90°** as expected:

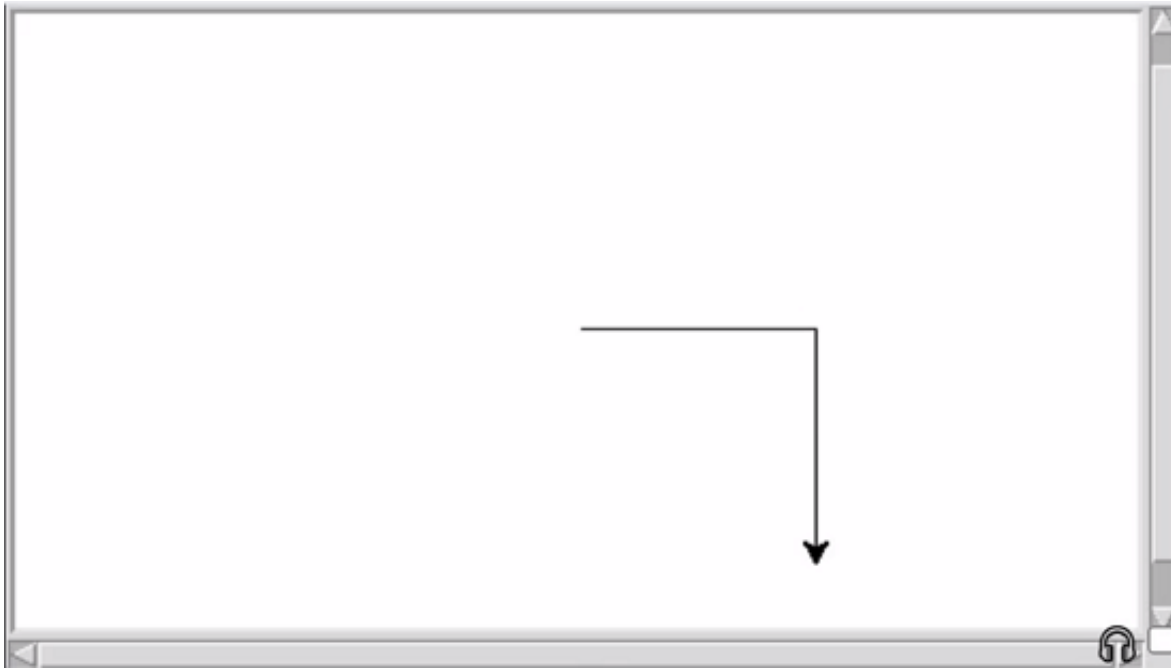


Note that you can also use the **left()** command to **turn left**. Check the **arrow** point for the **position** and **orientation** of the **turtle** after running your code.

Let's **add** the **forward** command **again** like so:

```
from turtle import *  
  
forward(100)  
right(90)  
forward(100)
```

Rerunning our code, we'll have:

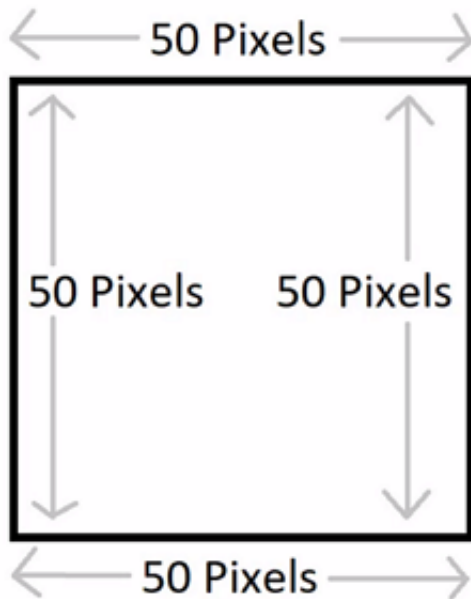


We see that once we **turned** the turtle to **its right**, its new **forward** direction changed to going **downwards**. In this sense, the turtle has executed the first forward command moving to our **right** 100 pixels, then **turned** right 90°, and executed the second forward command by moving **down** another 100 pixels.

This list of steps is the basis of all algorithms as we're giving the computer a set of instructions to follow and it's executing them in a specific order to get to the aimed result. That's why Python Turtle is a great tool for us as it's literally displaying the result in a graphic manner, where we can instantly see what our code is doing on the screen.

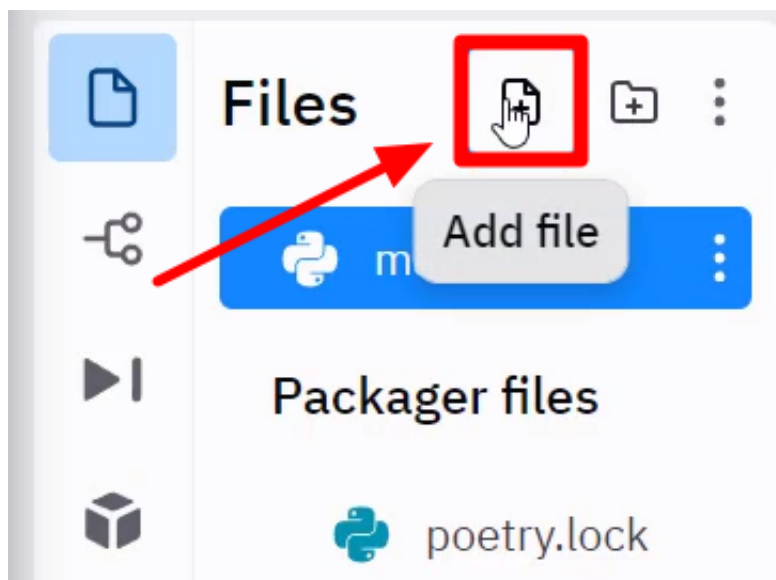
In this lesson, we're going to create a square with Python Turtle.

Try making a **50 by 50-pixel square** as shown below:



## Solution

Let's put our code so far in a **separate** file just to have everything organized. Click on the '**Add file**' button to create a **new** file:



Name your file ('**IntroLesson.py**', for instance) then **copy and paste** the code we have in **main.py** to this new file. **IntroLesson.py** is where we'll keep track of the code we've seen during our past lessons while **main.py** will have only the **current** code we're working on, as it is the file that gets to be run when we execute our project.

Let's start our solution code by **importing** the **Turtle** library, as otherwise none of the commands will work, and proceed by **moving forward** and **turning to the right** to draw the **top** side of the

square:

```
from turtle import *
```

```
forward(50)
```

```
right(90)
```

Similarly, **repeat** this same process **twice** more to draw the **right** and **bottom** sides:

```
from turtle import *
```

```
forward(50)
```

```
right(90)
```

```
forward(50)
```

```
right(90)
```

```
forward(50)
```

```
right(90)
```

Now we just need to get back to our **original position** to have the **left** side of the square done. For that, move **forward** one last time:

```
from turtle import *
```

```
forward(50)
```

```
right(90)
```

```
forward(50)
```

```
right(90)
```

```
forward(50)
```

```
right(90)
```

```
forward(50)
```

Click '**Run**' and there's our square all done:



## Pseudocode

[Pseudocode](#) is a step in the design process that can help us with creating algorithms. Like with flowcharts, pseudocode allows us to plan out the steps and order of execution before jumping into the code. The difference between a flowchart though, is that pseudocode is written like code, yet in English. It's basically like explaining to someone what each line of code will be doing, not needing to worry about the specific symbols or commands that Python requires.

So let's look at designing our square challenge with pseudocode!

First, we need to import Turtle.

```
import turtle
```

You'll see that I didn't write down *from turtle import \**. That's because we aren't coding with pseudocode, we're simply describing the algorithm.

Then to move forwards, we can write:

```
import turtle

move forwards 50 pixels
```

That line, specifies that we need to move our turtle forward 50 pixels. So when we get to the actual coding stage, we know exactly what we need to do. Here's what we could write for turning the turtle:

```
import turtle

move forwards 50 pixels
turn right 90 degrees
```

Again, specifying what we need to do. Now let's complete the pseudocode to create the entire square:

```
import turtle

move forwards 50 pixels
turn right 90 degrees

move forwards 50 pixels
turn right 90 degrees

move forwards 50 pixels
turn right 90 degrees

move forwards 50 pixels
```





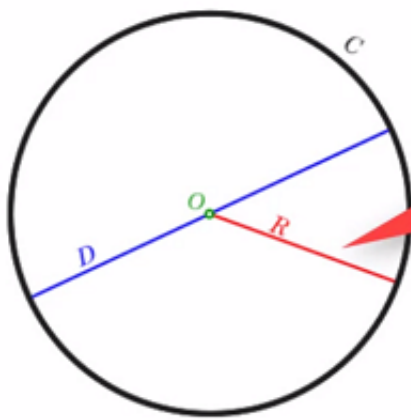
You can see how pseudocode can help you out (especially if you have a complex algorithm), as you're basically writing the algorithm beforehand. Another benefit of pseudocode, is that you can share it with other people that may not know your programming language.

In this lesson, we'll learn how to create circles with Python Turtle.

To **create** a **circle** we don't need to specify a series of commands, we need just **one**:

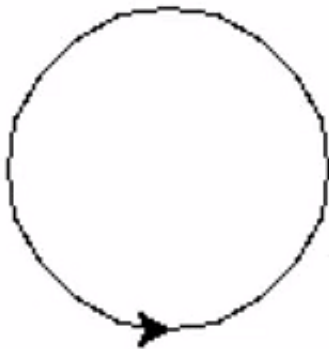
```
from turtle import *  
  
circle(50)
```

Between parentheses, we're passing in the **radius** of the circle:



Distance from the  
center of the circle  
to the edge.

Click 'Run':



We can see that the turtle is pointing to our right and it has moved forward and to the **left** in order to draw the circle, before getting back to the starting point.

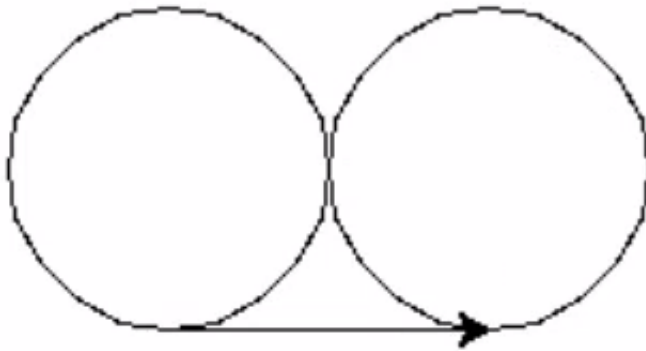
Let's **draw** another **circle** to the right of this first one by moving **forward** to the **right** then drawing the circle as follows:

```
from turtle import *
```



```
circle(50)
forward(100)
circle(50)
```

**Running** our code again:



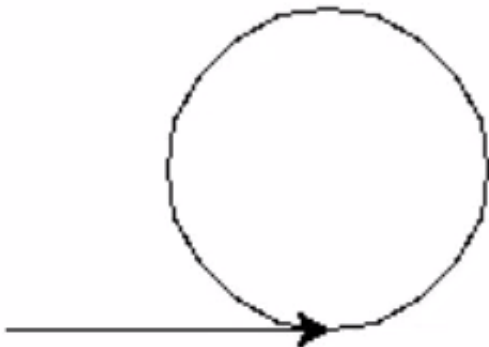
It's important to emphasize here how the **order** of the commands significantly alters the result we get given that if we **move** forwards **before** drawing the **circles**:

```
from turtle import *

forward(100)
circle(50)

circle(50)
```

We end up having one circle drawn **on top** of the other and the final outcome **differs** from what we intended:



**Instructions:**

Create a pattern or image using at least three different shapes and three different colors. You can use circles, squares, triangles, or any combination of shapes you've learned so far. Arrange them in any way you like to create a unique and colorful piece of art.

**Tips:**

- Use the `color()` function to change the pen color before drawing each shape.
- Remember that you can use hex color codes for a wider range of colors.
- Consider using `begin_fill()` and `end_fill()` to fill your shapes with color.
- Be creative with the arrangement—think about making a face, a landscape, or an abstract design.

In this lesson, we'll go over how to use colors with Python Turtle.

All commands we've seen in the previous lessons had our turtle leaving a simple **black trace** behind it so we could take note of its trajectory. To **change** the **color** of our output, we need to call for the color command **before** executing the desired action:

```
from turtle import *  
  
color("red")  
forward(100)
```

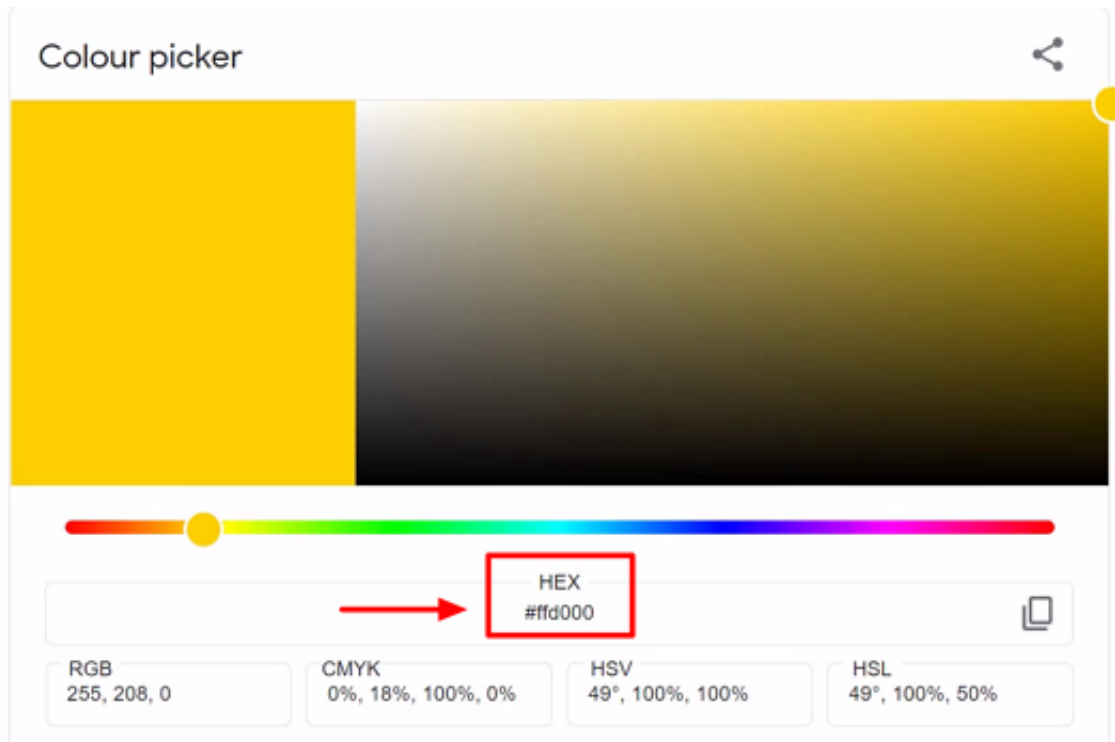
To enter the color we can **write** the **name** of the **color** in between **quotation marks**, and here we have it:



If we **invert** these two lines of code, we'll see that it **first** draws the **line** to the right in **black** and only then **changes** its color to **red**:



Feel free to use other colors as you like, there are also **light** shades that you can pick such as "**lightblue**". We can add **hex** colors to our command as well if we want more **specific** colors. For that, you can **search** for "**color picker**" in **Google** and you should be able to **choose** a color and get its **correspondent** hex value:



Now **copy and paste** it in our code to have this exact **golden color** reproduced in your program:

```
from turtle import *  
  
color("#ffd000")  
forward(100)
```



Notice that the **hex** colors **start** with the '#' symbol so that the computer is able to identify them correctly and also do make sure to always enter the color between quotation marks.

Moreover, we can **change** the colors more times **throughout** the code as needed:

```
from turtle import *  
  
color("#ffd000")  
forward(100)  
color("#00ff48")  
right(90)  
forward(100)
```



We're **initially** setting the color to be **gold** then we change it to **green** later on:



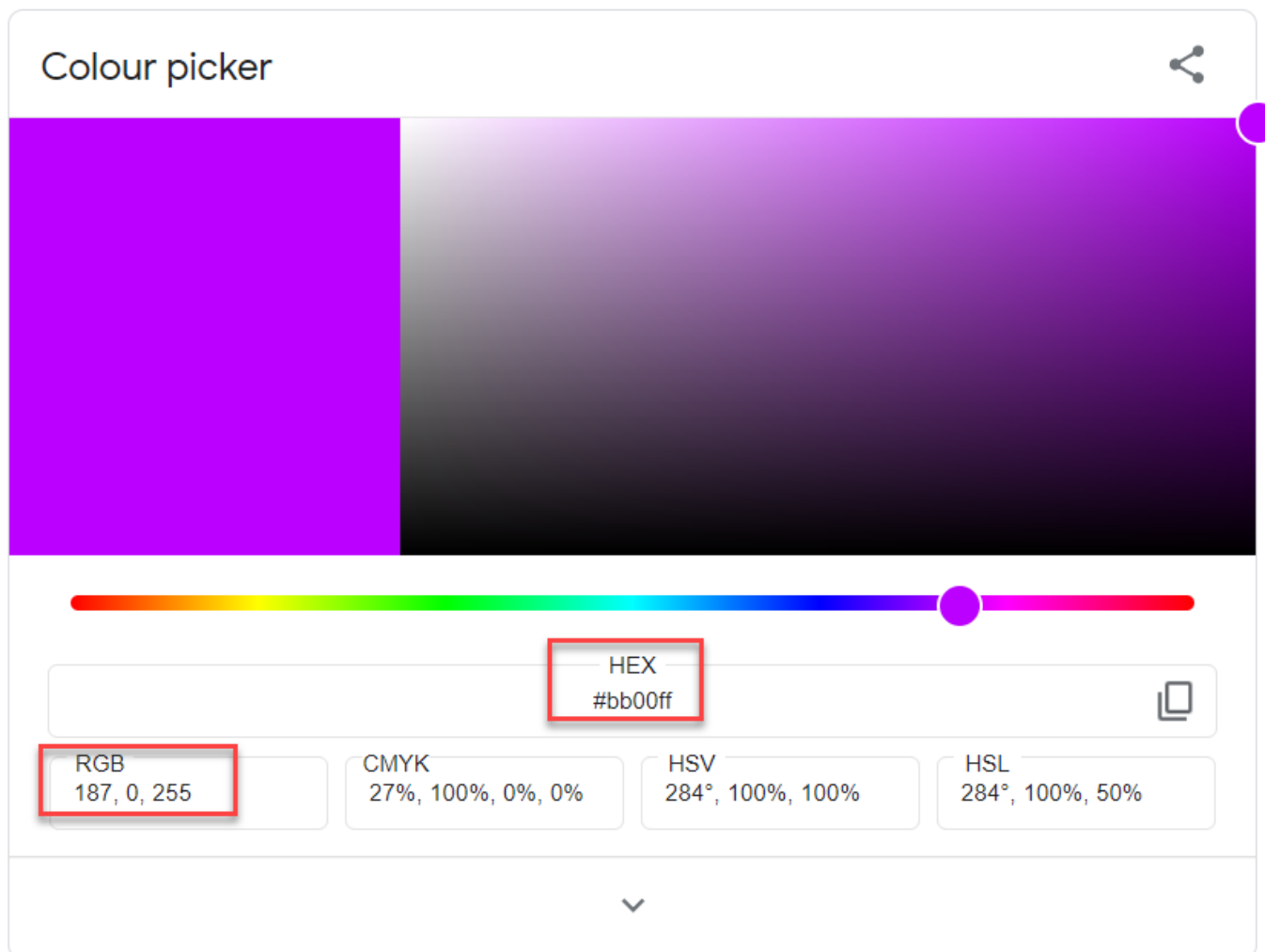
## Hexadecimal Colors

In our everyday lives, we count and think of numbers in the [decimal numeral system](#) (aka base 10). Base 10, meaning that we count in sets of 10. 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, then 11, 12, etc. Every 10 numbers we split them up (because we have 10 fingers). Computers on the hardware level use base 2, with 1's and 0's.

[Hexadecimal](#) is base 16. A benefit of this is that we can compress more numbers into a smaller representation. For example: 500,000,000 in hexadecimal is *1DCD6500*. You can see how it requires fewer characters to represent that number, thus saving on memory.

Since colors are represented on a computer with red, green, and blue values – it's very handy to compress that down to a single piece of text.

For example, the color *purple* can be represented as *#bb00ff* in hex. You can see below, how the hex value requires fewer characters than the RGB value.



Colour picker

RGB  
187, 0, 255

CMYK  
27%, 100%, 0%, 0%

HSV  
284°, 100%, 100%

HSL  
284°, 100%, 50%

HEX  
#bb00ff





## Challenge - Changing Color

As a challenge, create *three* circles – one green, one red, and one purple.

Once done, check out the next lesson for the answer.

In this lesson, we'll be looking at filling in our objects with color.

If you use any graphics editor, such as Paint or Photoshop, you're familiar with the **bucket tool** employed to fill in shapes with a chosen color.

Up until now, we've been drawing just the **outline** of objects. To **fill in** our objects with **color** using Python Turtle, we need to **encapsulate** everything we want to color with the **begin\_fill** and **end\_fill** commands:

```
from turtle import *  
  
begin_fill()  
circle(50)  
end_fill()
```

Here, we're drawing a **circle** that's going to be filled in with color. As the **default** color is **black**, we get a black circle as a result. Let's **add** the **color** command before our block of code to make the circle **red**:

```
from turtle import *  
  
color("red")  
  
begin_fill()  
circle(50)  
end_fill()
```

And here we have it:



If you go back to our square challenge from a past lesson, you can **replace** the circle command with the group of commands we used to create the **square** to create a **red** square as well:

```
from turtle import *  
  
color("red")  
  
begin_fill()
```

```
forward(50)
right(90)

forward(50)
right(90)

forward(50)
right(90)

forward(50)

end_fill()
```

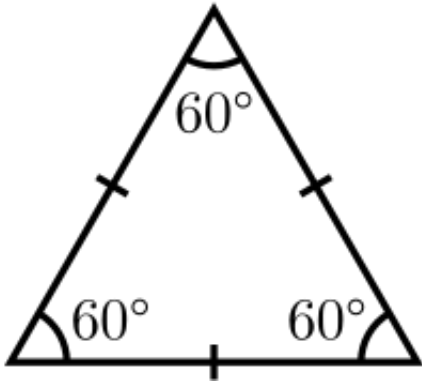
As a summary, `begin_fill` is used to denote that whatever shapes we draw next will be filled in with color until we enter the `end_fill` command. Note that when the computer runs the **end\_fill** command is when it actually **puts** the colors in the shapes that we have drawn since `begin_fill` was called, and then it **stops** filling in any objects from that point on in our code.



## Challenge - Filling in Shapes

As a bit of a challenge, draw and fill in an *equilateral triangle*.

An [equilateral triangle](#), is a triangle where all sides are of equal length, thus all of the angles must be the same too. For this challenge, make each side of the triangle *100 pixels*.



Once done, check out the next lesson for the answer.

In this lesson, we're going to have a look at how we can change the background color inside Turtle.

To **change** the **background color** we need to enter the **bgcolor** command passing in the **color** we want in between **quotation marks**:

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

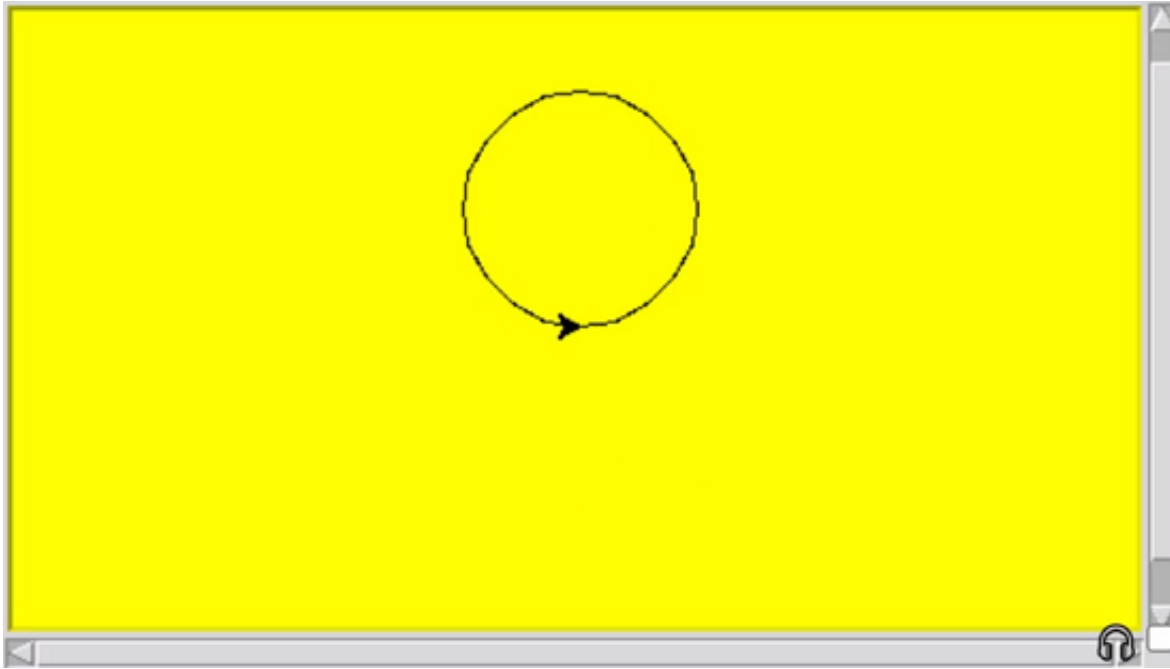
Now we have a **green** background instead of the white default one:



We can **draw** objects **on top** of our colored background normally:

```
from turtle import *  
  
bgcolor("yellow")  
  
circle(50)
```

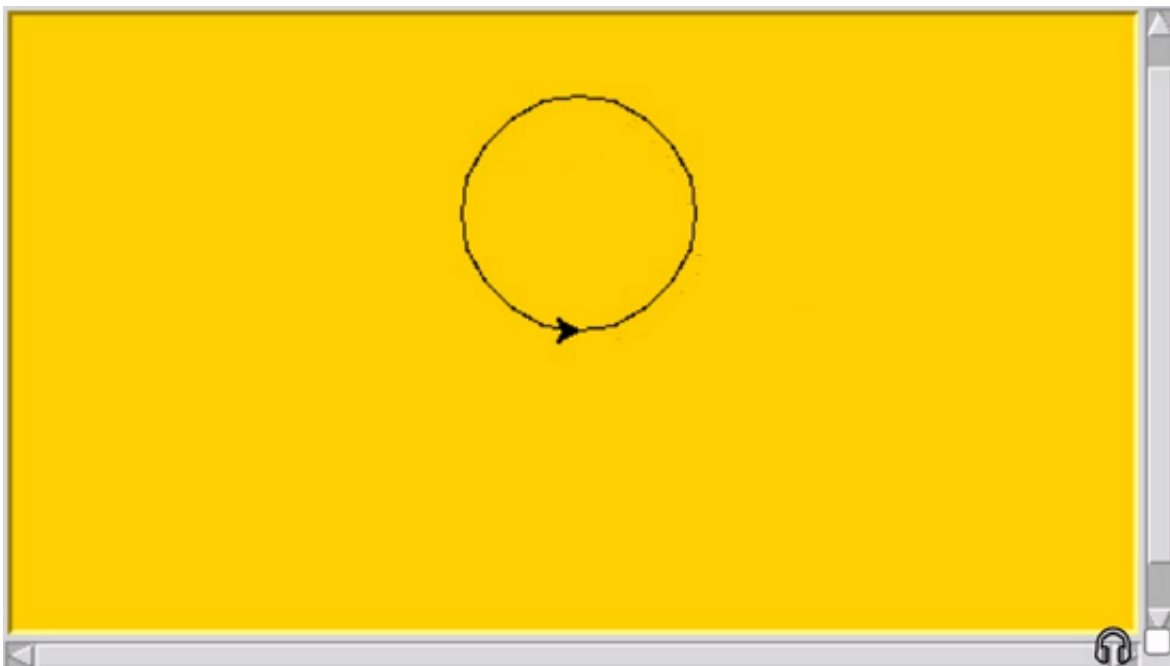
Running it, we'll have a **black circle** on top of a **yellow** background:



## Coloring the Background with Hex Colors

Just like we colored our objects with hex colors, we can also use them to color our background too:

```
from turtle import *  
  
bgcolor("#ffd000")  
  
circle(50)
```





In this lesson, we're going over the penup and pendown commands.

Up to this point, whenever we've **moved** our turtle it has left a **trail** behind, coloring all pixels it has passed over. Let's set up a situation where we don't want that to happen, for instance when we have **two circles** drawn close to each other:

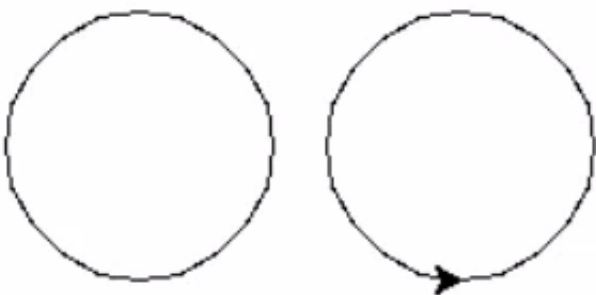
```
from turtle import *  
  
circle(50)  
forward(120)  
circle(50)
```

If we want no line connecting them together, in other words, we want to move the turtle **without** leaving a trace we can do that by using two commands: **penup** and **pendown**. Penup can be associated with **lifting** the pen up from a piece of paper, thus **stopping** to mark the paper with ink, while pendown corresponds to getting back to **writing** on the paper.

To **erase** any **trails** from our code above using these 2 commands, we can do as follows:

```
from turtle import *  
  
circle(50)  
penup()  
forward(120)  
pendown()  
circle(50)
```

We see that now all that's in **between** penup and pendown is **not** being drawn to the screen:



**Instructions:**

Using filled shapes, draw a simple house. Your house should have at least:

- A square or rectangle for the main building.
- A triangle for the roof.
- At least one window or door, using any shape.

Feel free to add additional details like a chimney, pathway, or surrounding trees using shapes and colors.

**Tips:**

- Use `begin_fill()` and `end_fill()` to fill in each shape.
- Change colors between shapes to make your house more vibrant.
- Use `penup()` and `pendown()` to move between different parts of your drawing without drawing lines.



In this lesson, we'll start to plan out our solar system project using Python Turtle.

In terms of **project management**, we'll go over the following steps:

Step	Meaning
<b>Definition</b>	List of all the criteria we need to meet for this project
<b>Design</b>	Creation of a diagram, flowchart, and pseudocode for a better understanding of our project
<b>Implementation</b>	Coding the project in Python
<b>Evaluation</b>	Final overview of what's been done

## Definition

We're going to create a **solar system** with **Python Turtle** that has a **black background** and different **colored planets**:



This project serves as an example of how we can use **Python** to create **digital art** with the tools we've learned in the course.

In the next lesson, we'll continue with the designing process of what we'll be creating.

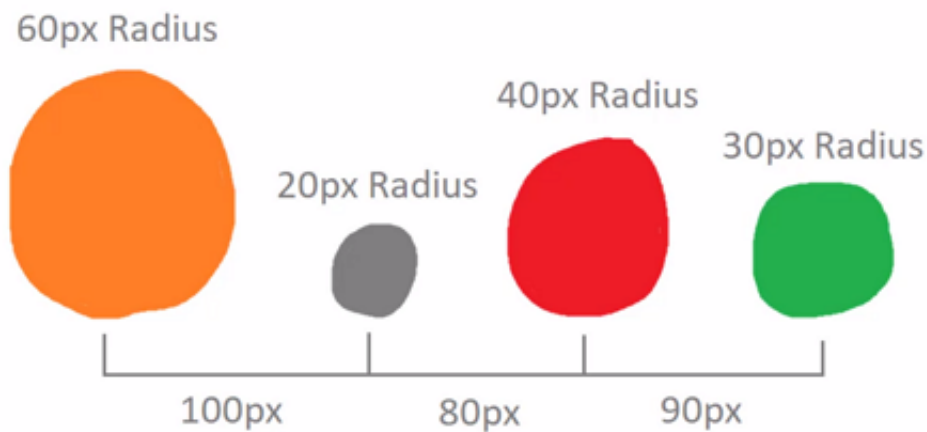
In this lesson, we're getting started with the design of our project.

We'll go over **3 different elements** that'll help us understand what our project will look like, they being:

- **Diagram** - Gives us a visual representation of the final outcome
- **Flowchart** - Dictates the flow and structure of our algorithm
- **Pseudocode** - Simplified version of the final code of our project

## Diagram

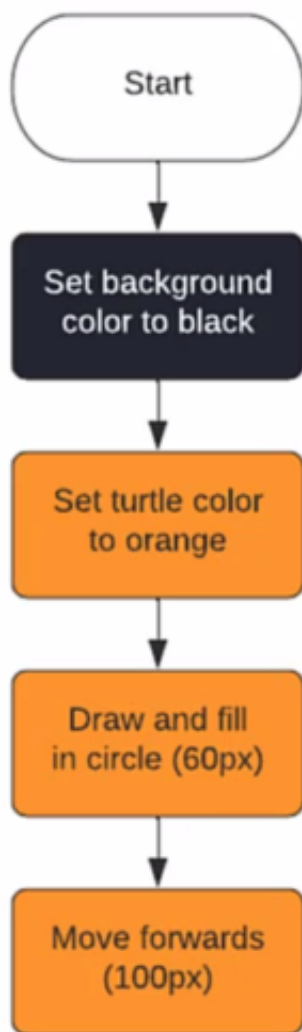
To make our diagram, we **start** off by drawing the **colored planets** we want to create, then we **add** extra details in order to have a better notion of how **big** each planet is and how **distant** they are from each other:



Note that anyone following the specifications of this diagram should be able to **produce** basically the **same** result, as it contains all the needed information to achieve the desired outcome.

## Flowchart

We begin our flowchart by **setting** the **background color** to **black**. Next, we set the **color** of the **turtle** to **orange** in order to **draw** and **fill in** our first **planet**. After that, we **move forwards** so that we're ready to draw the next planet:

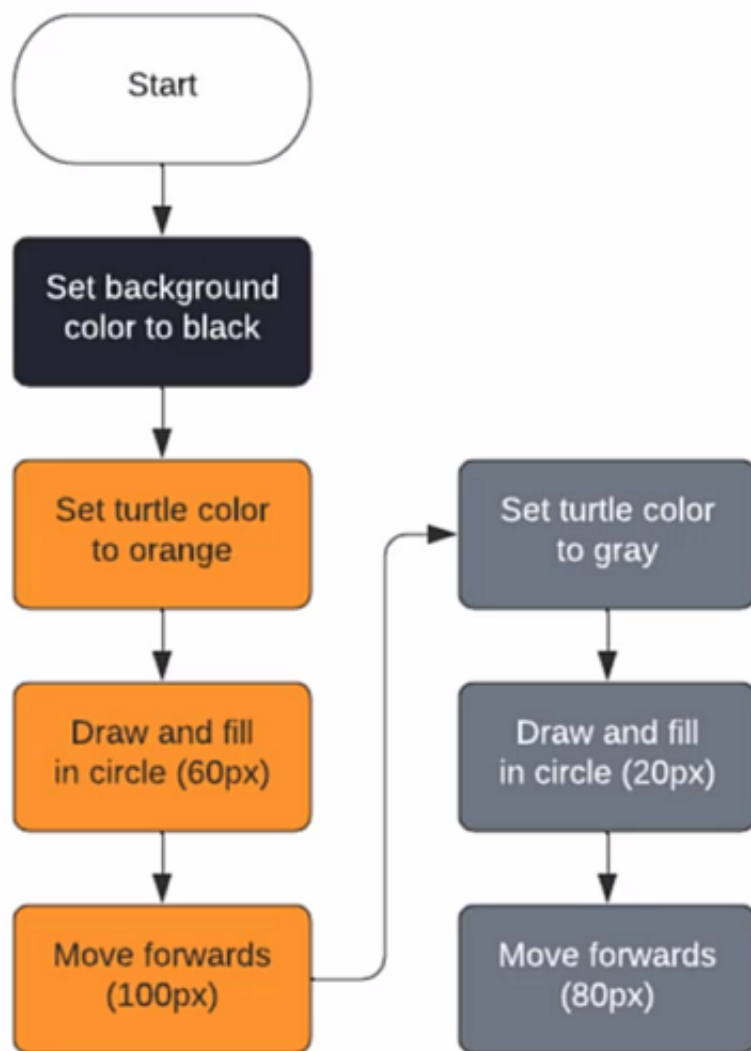


60px Radius



The **arrow** below the flowchart represents our **turtle's position** and **color**.

**Repeat** the same process for the **gray planet**:



60px Radius



20px Radius



Now, try **filling out** the rest of the **flowchart** then check the **final version** below:



The sequence of steps for all planets is very similar, changing only for the **last** planet where there's no need for us to move the turtle no more.

In the next lesson, we'll look at the correspondent pseudocode for the diagram and flowchart we've just created.

In this lesson, we're moving on to the last step of our design phase which is the pseudocode.

**Pseudocode** is an **informal** way to **write down** an **algorithm** in a way that's easier for humans to **understand**. As such, we write it out in **English** and not directly with the equivalent code in Python.

Let's take our **flowchart** as a base for our pseudocode:



We can go over the steps of the flowchart to **write** them down as a set of **instructions** for a recipe:

START

Set background color to BLACK

Set turtle color to ORANGE

Begin fill  
Draw circle  
End fill

Lift pen up  
Move forwards  
Put pen down



Remember that we need both '**begin fill**' and '**end fill**' commands in Turtle to fill our objects with color. Also, before moving the turtle, we want to **lift up** our pen not to leave any traces of ink connecting the planets, **putting it down** again after we've moved.

As a **challenge** try finishing up the rest of the steps of our pseudocode and then check the **complete** version below:

START

Set background color to BLACK

Set turtle color to ORANGE

Begin fill  
Draw circle  
End fill

Lift pen up  
Move forwards  
Put pen down

Set turtle color to GRAY

Begin fill  
Draw circle  
End fill

Lift pen up  
Move forwards  
Put pen down

Set turtle color to RED

Begin fill  
Draw circle  
End fill

Lift pen up  
Move forwards  
Put pen down

Set turtle color to GREEN

Begin fill  
Draw circle  
End fill

END

Here, we're **enclosing** the algorithm with the “**Start**” and “**End**” statements, just like we did in the flowchart as well.

In the next lesson, we're taking this pseudocode to its actual implementation with Python.

In this lesson, we're going to begin the implementation phase of our solar system project.

Following the list of instructions of our flowchart and pseudocode, let's **start coding**:

```
from turtle import *  
  
bgcolor("black")  
  
color("orange")  
begin_fill()  
circle(60)  
end_fill()
```

Here, we have our **background** color set to **black** and our first **planet** drawn and filled in with **orange**. As it's very important to continuously check that everything is working as expected, let's run our program to see how it's going so far:



Let us now **move forwards** a **100** pixels:

```
from turtle import *  
  
bgcolor("black")  
  
color("orange")  
begin_fill()  
circle(60)  
end_fill()  
  
penup()  
forward(100)  
pendown()
```

Run it again:





Next, repeat the process above to **draw** the **gray** planet:

```
from turtle import *

bgcolor("black")

color("orange")
begin_fill()
circle(60)
end_fill()

penup()
forward(100)
pendown()

color("gray")
begin_fill()
circle(20)
end_fill()
```

And so we have:



## Speeding Up the Animation

You may notice that it does take some time for the animation to run, and if you're testing it frequently you'll want it to run **faster**. A way to improve on that is by using the **speed command** to change the speed of the drawing animation.

At the **top** of your code, right **after** importing Turtle, write:

```
speed(0)
```

The **parameter** for this command is a **number** between 0 and 10. We're passing **0** for the animation to run **instantly**. If you pass **1** the animation will run the **slowest** and **10** will run the animation its fastest while still showing all the steps being performed by the program.

In this lesson, we're going to conclude our code for the solar system project.

## Adding Comments

As this project is going to be larger than our previous codes, it's a good practice to add **comments** to the code. Essentially, they are just notes that you can add to the code containing **explanations** or **details** about specific parts of the code. To do it, start with the **hashtag** symbol and then add your comment:

```
from turtle import *

speed(0)

bgcolor("black")

# create the ORANGE planet
color("orange")
begin_fill()
circle(60)
end_fill()

# move forwards
penup()
forward(100)
pendown()

# create the GRAY planet
color("gray")
begin_fill()
circle(20)
end_fill()
```

Note how comments appear in **gray** in the code editor as a way to easily spot them, and if you remove the hashtag you'll have an error when running your code as it's what is marking a particular line of code as a comment and not a command. Comments are **not run** by the computer as is the rest of the code, they're simply there to help us **organize** things or catch up to other people's code.

## Completing the Code

Try **proceeding** with the code for the **last 2 planets**, then check the final version below:

```
from turtle import *

speed(0)

bgcolor("black")

# create the ORANGE planet
color("orange")
begin_fill()
circle(60)
end_fill()
```

```
# move forwards
penup()
forward(100)
pendown()

# create the GRAY planet
color("gray")
begin_fill()
circle(20)
end_fill()

# move forwards
penup()
forward(80)
pendown()

# create the RED planet
color("red")
begin_fill()
circle(40)
end_fill()

# move forwards
penup()
forward(90)
pendown()

# create the GREEN planet
color("green")
begin_fill()
circle(30)
end_fill()
```

This is the **final view** of our project:



In this lesson, we're going over the evaluation of our project.

As the final stage of our project management, we need to **evaluate** if we **met** the **criteria** that we had defined at the start which were:

- Creation of a solar system using **Python Turtle**
- Contains a **black background** and different **colored planets**
- Developed using **tools** we've **learned** so far
- It's an example of how we can use Python to create **digital art**

As can be seen, we've **successfully covered** all these topics as we implemented our code.

So that concludes our solar system project for the course! Feel free to further expand on this as you wish.

**Instructions:**

Design and draw your own planetary system. Include at least:

- A star (circular shape).
- Four or more planets of different sizes and colors.
- Optional: Moons, asteroids, rings around the planets, or any other element you'd like to include!

Arrange your planets in any configuration you like—they don't have to be in a straight line.

**Tips:**

- Use `penup()` and `pendown()` to position your planets without drawing lines between them.
- Experiment with different `circle()` sizes to represent various planet sizes.
- Get creative with colors and positioning to make your planetary system unique.

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.

## IntroLesson.py

### Found in the Project root folder

This Python code uses the turtle graphics module to draw the shape of a right angle. It moves the turtle forward by 100 pixels, rotates it 90 degrees to the right, and again moves it forward by 100 pixels.

```
from turtle import *  
  
forward(100)  
right(90)  
forward(100)
```

## BackgroundColor.py

### Found in the Project root folder

This code uses Python's turtle graphics module to create a window with a gold background, and draw a circle with a default black outline and a radius of 50 pixels.

```
from turtle import *  
  
bgcolor("#ffd000")  
circle(50)
```

## Circles.py

### Found in the Project root folder

This Python code uses the turtle graphics module to draw two circles. It first draws a circle with a 50 pixel radius, then moves the turtle 100 pixels forward, and finally draws a second circle of the same size.

```
from turtle import *  
  
circle(50)  
forward(100)  
circle(50)
```

## Color.py

### Found in the Project root folder

This Python code uses the turtle graphics module to draw two lines at a right angle. It first sets the color to gold and draws a line 100 pixels forward. Then it changes the color to bright green, turns the turtle 90 degrees to the right, and draws another 100 pixels long line.

```
from turtle import *

color("#ffd000")
forward(100)
color("#00ff48")
right(90)
forward(100)
```

### Filling.py

#### Found in the Project root folder

This Python code uses the turtle graphics module to draw a filled circle. It sets the pen color to red, starts the filling process, draws a circle with a radius of 50 pixels, and completes the filling, resulting in a red filled circle.

```
from turtle import *

color("red")

begin_fill()
circle(50)
end_fill()
```

### LiftingThePen.py

#### Found in the Project root folder

This Python code uses the turtle graphics module to draw two circles. It starts by creating a circle with a radius of 50 pixels. It then lifts the pen, moves the turtle forward by 100 pixels, lowers the pen, and draws a second circle with a radius of 20 pixels.

```
from turtle import *

circle(50)

penup()
forward(100)
pendown()

circle(20)
```

### SolarSystemProject.py



## Found in the Project root folder

This Python code uses the turtle graphics module to draw four circles in different colors representing planets on a black background. The drawn planets are in orange, gray, red, and green colors with varying radii. After each planet is drawn, the turtle moves down before drawing the next one.

```
from turtle import *

speed(0)

# set the background color
bgcolor("black")

# create the ORANGE planet
color("orange")
begin_fill()
circle(60)
end_fill()

# move down
right(90)
penup()
forward(70)
pendown()
left(90)

# create the GRAY planet
color("gray")
begin_fill()
circle(20)
end_fill()

# move down
right(90)
penup()
forward(100)
pendown()
left(90)

# create the RED planet
color("red")
begin_fill()
circle(40)
end_fill()

# move down
right(90)
penup()
forward(100)
pendown()
left(90)

# create the GREEN planet
color("green")
begin_fill()
```

```
circle(30)  
end_fill()
```

## SquareChallenge.py

### Found in the Project root folder

The Python code uses the turtle graphics module to draw a square. The turtle moves forward 50 pixels and then turns right by 90 degrees; this operation is repeated four times, resulting in a square shape.

```
from turtle import *  
  
forward(50)  
right(90)  
  
forward(50)  
right(90)  
  
forward(50)  
right(90)  
  
forward(50)
```