Programming Fundamentals

Introduction to Processing

Python Reference Manual (Version 1.0)

Made by Kainat Abbasi

Contents

1. Getting Started

1.1 What is Processing?

Processing is an integrated development environment (IDE) or more accurately a computer language for coding within the context of visual arts. This software is specifically designed for the purpose of production and prototyping, generation and modification of images useful for students, artists, design professionals and researchers. Processing initially integrated Java to allow programmers to write code, but today one may use python and JavaScript as well.

1.2 Installing Processing

You may install the software by visiting `http://processing.org/download` and selecting the Mac, Windows, or Linux version, depending on what machine you have. This is how installation works:

- On Windows, you'll have a .zip file. Double-click it, and drag the folder inside to a location on your hard disk. It could be Program Files or simply the desktop, but the important thing is for the processing folder to be pulled out of that .zip file. Then double-click processing.exe to start.
- The Mac OS X version is also a .zip file. Double-click it and drag the Processing icon to the Applications folder. If you're using someone else's machine and can't modify the Applications folder, just drag the application to the desktop. Then double-click the Processing icon to start.
- The Linux version is a .tar.gz file, which should be familiar to most Linux users. Download the file to your home directory, then open a terminal window, and type: tar xvfz processing-xxxx.tgz (Replace xxxx with the rest of the file's name, which is the version number.) This will create a folder named processing-2.0 or something similar. Then change to that directory: cd processing-xxxx and run it:

If the program doesn't start, or you're otherwise stuck, visit the troubleshooting page for possible solutions.

1.3 Let’s Begin

- When you begin working with the processing software, you will firstly see a large area which is the **text editor**. The text editor is used for coding, all of the programming codes and instructions are entered in the text editor.
- A row of buttons across the top is the **tool bar**.
- Below the editor is the **Message Area**, which is used for one line messages. The message area reports an error if there is an error in your program. Your program will not run unless you correct those errors.
- Below the message area is the **console**. The `print()` and `println()` functions write text to the console which enables the user to display a variable or confirm an event. You may also print any messages that you might have written for your understanding of the code or print the output of some mathematical expression that your code has to evaluate.
- There is the **Run button** in the tool bar (triangle). When you click the Run button your program will be compiled and executed if there are no errors.
  If you didn’t type your code correctly or if there is an error in your program, the Message Area will turn red and complain about the error. If this happens, go through your code again and make the required changes.

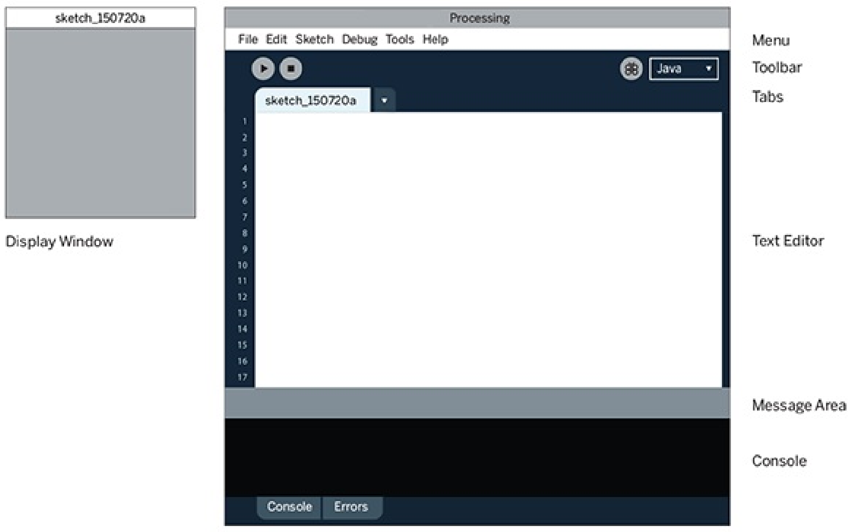


Figure 1.1: The Processing Window

Make sure that:

- the numbers(parameters that you give to your functions) are contained within parentheses
- have commas between them as you’d see in the next section.

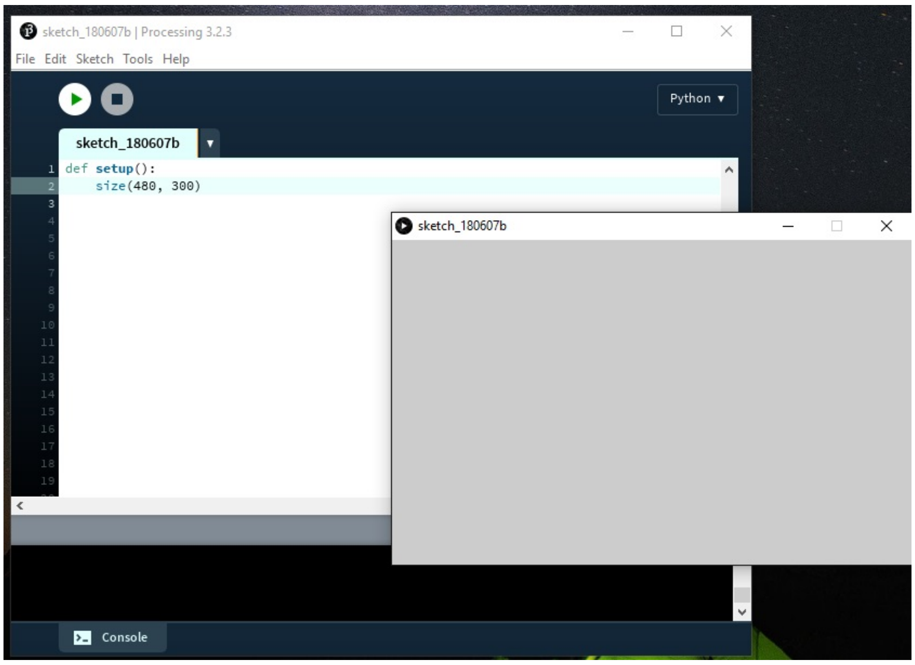
1.4 Syntax

This is our first code

```
def setup():
   size(480, 120)
def draw():
    if mousePressed :
        fill(0)
    else:
        fill(255)
    ellipse(mouseX , mouseY , 80, 80)
```

1.4.1 setup()

A **setup**() function is where you control the visualization properties like screen size. A very basic setup() function will set up the processing window size using the `size` function. `size` takes 2 parameters, the first one specifies the width of the window and the second one specifies the height.


sketch_180607b | Processing 3.2.3
File Edit Sketch Tools Help
Python
sketch_180607b
sketch_180607b
Console

```
def setup():
    size(480, 300)
```

Figure 1.2: The Processing Window

The following code for example, will only setup the processing window for you which will be 500 pixels wide and high.

■ **Example 1.1** Python Example Code

```
def setup():
    size(500,500)
```
■

`def` is the keyword for initiating any function in Python, which will always be followed by the name of your function, parenthesis and a colon as shown in example 1.1. The name of your function can be any name of your choice except in processing, where the `setup()` and the `draw()` functions are always used to set your window size and run animations respectively. Apart from these, you can initiate any other function and give it a name of your choice. Functions and their syntax will be covered in detail in chapter 4.

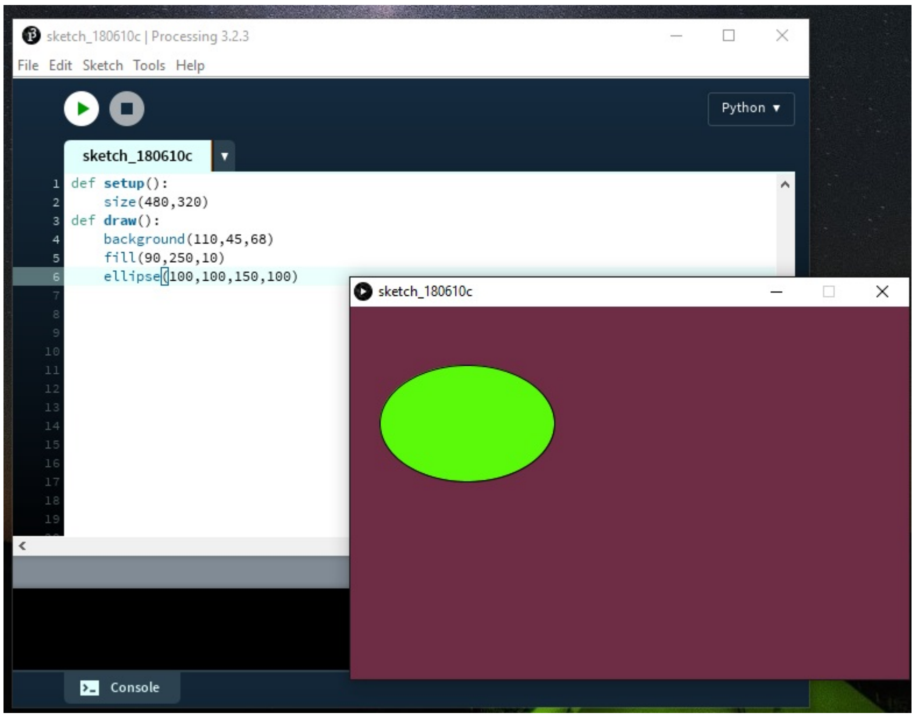
1.4.2 draw()

The draw() function is called immediately after setup(). After setting the size of the drawing window, other functions including shapes such as line(x1,y1,x2,y2) or rect(x1,y1,x2,y2) or code

related to animation are all included in the draw function. Functions such as background() which specify the background of the screen are included in draw() if they are to be updated while the program is running otherwise you may include them in the setup() function as well.
Lines of code included in draw() are executed continuously i.e your program will keep executing those lines again and again unless you close your program or use the noLoop() function which is used to prevent your code from running continuously in the draw() function.

The processing window

Your processing window is basically the Cartesian coordinate system. A position on the display window in processing is specified with respect to the fact that the upper left corner of your screen is the point (0,0) by default. The x and y coordinates of your shapes will specify their position on the screen as all the shapes take the x and y coordinates of their position as parameters as you will see in the following example.


sketch_180610c | Processing 3.2.3
File Edit Sketch Tools Help
Python ▾
sketch_180610c
sketch_180610c
Console

```
def setup():
    size(480,320)
def draw():
    background(110,45,68)
    fill(90,250,10)
    ellipse(100,100,150,100)
```

Figure 1.3: Example Code

The **ellipse** function used here is used to construct an ellipse on your window. It takes four parameters, two of which are the x and y coordinates of the center of your ellipse while the other two specify the height and the width. You can use the same function to construct a circle if both height and the width are equal.

1.4.3 Basic Shapes

A few shapes are listed below.

point()

A point function draws a point on your screen and takes two parameters that specify its position.
Syntax :`point(x, y)`

line()

Takes 4 parameters, 2 of which specify the x and y coordinates of the starting point of your line while the other two specify the x and y coordinates of the end point of your line.
Syntax :`line(x1, y1, x2, y2)`

triangle()

Takes six parameters which are x and y coordinates of all of its vertices.
Syntax : `triangle(x1, y1, x2, y2, x3, y3)`

rect()

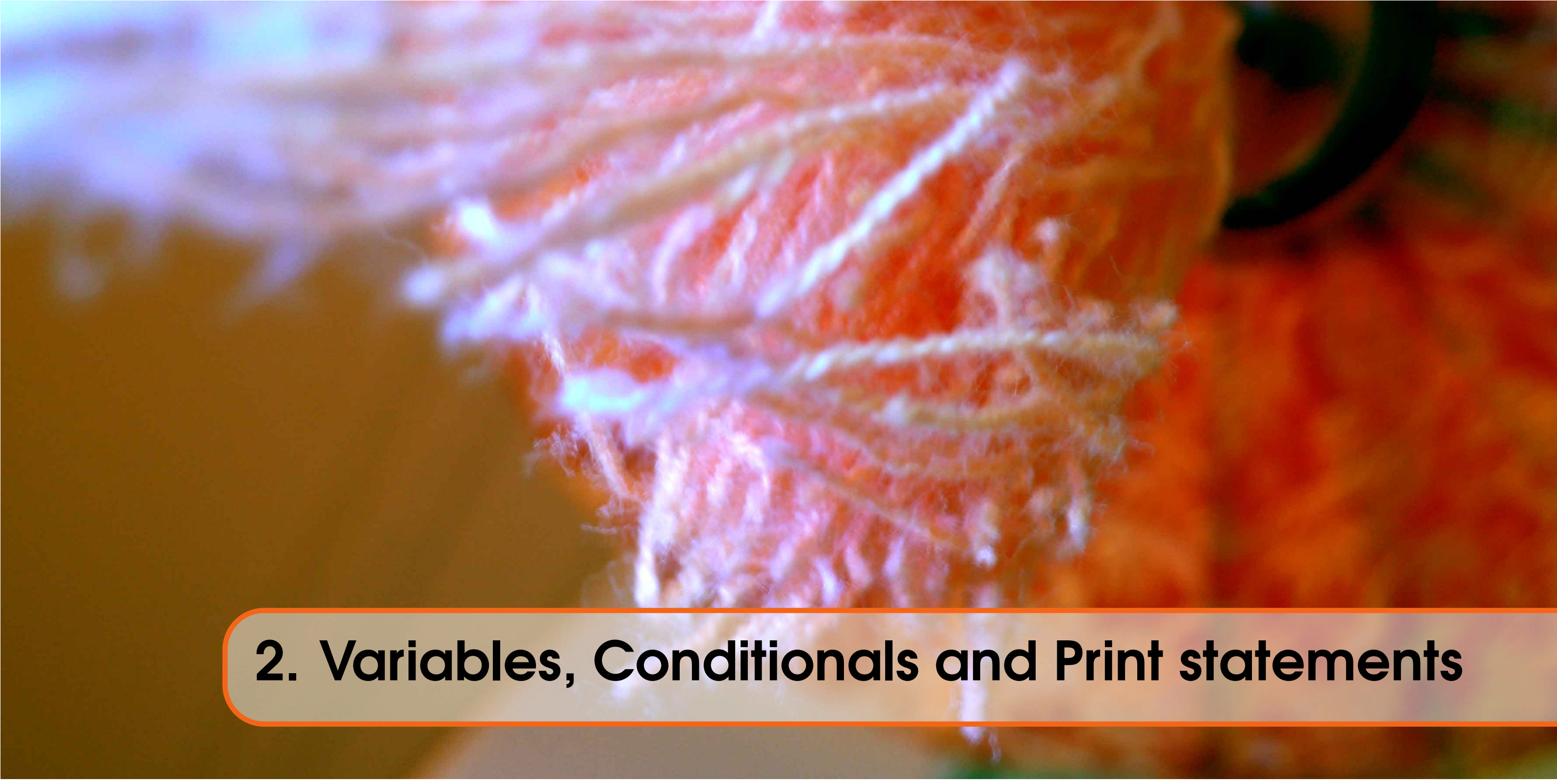
Takes 4 parameters. The first two parameters specify the x and y coordinates of the upper right corner of your rectangle, while the other two specify the height and width.
Syntax : `rect(x, y, height, width)`

quad()

Takes 8 parameters which specify the x and y coordinates of all four sides of your quadrilateral.
Syntax: `quad(x1, y1, x2, y2, x3, y3, x4, y4)`

1.5 Exercises

1. *Draw a display window 500 pixels high and 250 pixels wide. Look up the background() function and use it to give a color to your window.*
2. *For the same window, draw a circle and give it a red color. Draw a rectangle inside the circle and give it a different color from that of the background and the circle using RGB values.*
3. *Use the quad() function to construct a trapezium exactly at the center of your window.*



2. Variables, Conditionals and Print statements

2.1 Variables

A variable stores data and allows that data to be used several times in a program. For example, if we write x=23, then x is the variable and 3 is the value it is assigned. Every time you refer to x, it will be replaced with 23 or read as 23 by your machine. Variables can be updated, e.g., in a program if you reassign x a value, it will be updated to the new value and the older one will no longer be valid. Your program will continue with the new value of x.

■ **Example 2.1** Python Example Code

```
x = "Keep up the spirits, seven more chapters to go!" # assigns x a string
print(x) #prints the value of x on the console
```

Output:

```
Keep up the spirits, seven more chapters to go!
```
■

Here is another example:

■ **Example 2.2** Python Example Code

```
def sum():
x = 3
y = 4
output = x + y  #Evaluates the sum of the variables x and y and stores them in
another variable i.e.  output
print(output)
```

Output:

```
» 7
```
■

In every computer program, it is important to declare the variable before it is used. In Python, you will also have to assign an initial value to the variable. You may specify the data type of the variable if you want your program to store the value in a specific data type. Data types are further explained in the coming sections.

2.1.1 Global and Local Variables

While programming in Python, it is the choice of a programmer to either declare a variable inside a function or outside. Any variable that is declared outside a function can be accessed and used inside any function by declaring it global within the function.
On the other hand, if a variable is declared inside a function, no other function has access to that variable and the variable exists only for that specific function. Such a variable is a local variable.

■ **Example 2.3** Python Example Code

```
 def setup():
    size(500,500)
def draw():
    x = 1 (x is assigned a value of 1)
    background(x)
    x = x + 1 (x now becomes 2)‘
```
■

For example, in example 2.3, x is a local variable as it is declared in the local scope, i.e., inside the draw() function and thus, no other function can access it. Here is how the code works:
Example 2.3 sets the background of your processing window to the value of x. Line # 6 increments the value of x by one every time the code repeats, i.e, when your code runs the second time the value of x should become 2, for the third time it should become 3 and so on, but since line # 4 assigns x a value of 1 inside the draw function, therefore whenever the code repeats, your value changes back to 1 and the value of x just switches from 1 to 2 and back to 1. In this way your background is always assigned a value of 1. Thus, x is a local variable whose value may change inside the function but you cannot access x outside the function.

If your value is to change while the program is repeating then it is placed outside any function and declared a global variable. A global variable is a variable that any function can access and The following code declares x a global variable.

■ **Example 2.4** Python Example Code

```
def setup():
    size(500,500)
    x = 0
def draw():
    global x
    background(x)
    x = x + 1
    print(x)
```
■

Since x is a global variable it can be accessed and used by any function. The draw() function here uses the value of x to set the background of your drawing window.

2.1.2 Data types

Processing allows you to save a range of data including numbers, alphabets, images, colors, font etc. Computer system stores data in bits. Every kind of data whether a number or alphabet, is stored in form of 1s or 0s making it necessary for your system to differentiate between the two. Different values are stored differently, for e.g numbers and boolean values are two separate data types and are both stored separately from each. While storing a value, if you specify an incorrect data type, your system will either return an error or convert the data type into the one specified. Data types tell your system about the kind of value you are storing. For e.g, an integer data type stores whole numbers while float stores decimal numbers. There are several data types:

- ***int:*** It is the integer data type and stores numbers that do not have a fractional part, i.e., whole numbers. The data type is specified at the time of storing a variable and is written with the value. For e.g `x = int(2.345)` will store 2 in the variable x because 2.345 is not an integer, and your system converts your value to the data type specified.

  *It is not important to specify the data type in **Python while declaring or assigning a variable**. For all other languages your program will return an error if the data type is not specified. For example, x = 2 will automatically be stored as an integer and x = 2.5 will be stored as float in Python. It is not necessary to say x = int(2) for Python to store it as an integer.*

- ***float:*** It is used to store decimal numbers or numbers that have a fractional part, for example, 10.05 or 0.0 are both floating point numbers. If you store an integer value in the float data type your system will store it as a float value with a decimal part. For e.g `x = float(1)` will be stored as 1.0 in your program.

- ***char:*** This data type contains *single* alphabets, symbols or other specialized codes. Any variable that stores only one character is included in the type `char`. Every possible value in a char data type is a character code in the Unicode character set. ASCII characters are also included in the Unicode character set.
  There is no `char` data type in Python because single characters are also stored as strings in your program.
  Read more about ASCII characters at `http://www.asciitable.com/`

- ***bool:*** The `bool` or boolean data type stores only two values either True or False. This data type is important in deciding which lines of code to run. The boolean value False is the integer value 0, whereas True can be any integer value(usually 1) other than 0.

- ***Strings:*** A sequence of characters is stored in the string data type. For example, `"Processing"` is a string data type consisting of a sequence of alphabets. Strings are stored using double-quotes. For some languages(including python), single quotes are also used to store strings. For most of the other languages single quotes are used to store `char`, while double quotes store strings. Digits in double quotes are also included in the string data type and arithmetic operations can only be performed on them if they are converted to int or float data type.

■ **Example 2.5** Python Example Code

```
string1 = "This doesn't end here!  There are two complete chapters on Strings
in this manual."
x = '!'
print(x)
print(string1)
```

Output:

» !
» This doesn't end here! There are two complete chapters on Strings in this manual. ■

2.1.3 Arithmetic Operations

The processing window is basically the Cartesian coordinate system where you input numeric values to specify the position of a shape, line or anything you wish to draw. The dimensions of the shape decide their position, while some shapes take parameters such as width and height to decide the size of the shape. It thus, requires you to perform mathematical operations, for example, if you want to draw a rectangle that is half the size of your window, then you may simply divide the height and width of your window by 2 and input the output as parameters of your `rect()` function. This is what your code should look like:

■ **Example 2.6** Python Example Code

```
def setup():
    size(500,500)
def draw():
    rect(120,120,width/2,height/2)
```
■

In the code above, you did not declare any variable named `width` or `height`, but your program automatically did. The `size` function takes the values of height and width as parameters which is then stored in your program. In this code, the variables width and height store the value 500 each.

Simple arithmetic operations can help you change position or attributes of elements on your screen. For example, the + symbol is used for addition while the - symbol is used for subtraction. The following table lists common mathematical operators used in Python.

Operator	Operation	Example
+	Addition	3 + 4 » 7
−	Subtraction	3 - 4 » 1
∗	Multiplication	3 * 4 » 12
/	Division	4 / 3 » 1.333333
%	Modulus(returns the remainder)	4 % 3 » 1

In python // is used for floor division. Floor division will always return whole number value dropping all the numbers after the decimal point.

2.1.4 Arithmetic operations on different data types

Arithmetic operations on variables or numbers are performed according to their data types. For example, if you add two integers, your program will always return an integer value unless specified. Similarly, if you add two float values, your program will return a float value. One float value added to an integer will return a float value according to basic mathematic rules. However, if you specify the type of your values, your program will return the value in the desired data type.
The following examples will help you understand how different data types work.

The following code multiplies 2 float values and returns an integer value:

■ **Example 2.7** Python Example Code

```
x=3.5
y=9.0
print(int(x*y))
```
Output:

» 31 ■

*x*y will give a float value whereas int(x*y) will convert the float value into an integer and return an integer value.*

Exercise 2.1 Divide any two integer values using floor division and return a float value. ■

2.1.5 Common Operators

Logical operators applicable in python are and, or, and not. Operators are usually used in conditionals. You will learn more about them in the coming sections.
Other commonly used operators include assignment operator which is the equal to(=) symbol and is used to assign values to a variable. The comparison operator is the double equal to(==) symbol that compares if the two values are equal. Other operators are listed in the table below:

Operator	Operation	Example
==	Indicates equality. If two values are equal then the condition is True	"a" == "a" » True
! =	If two values are not equal, then the condition is True	3 != 4 » True
<	Left operand being less than the right operand, returns True	3 < 4 » True
>	Left operand being greater than the right operand, returns True	4 > 3 » True
<=	Left operand being less than or equal to the right operand, returns True	3 <= 4 » True
>=	Left operand being greater than or equal to the right operand, returns True	4 >= 3 » True

Read more about operators at `https://www.tutorialspoint.com/python/python_basic_operators.htm`

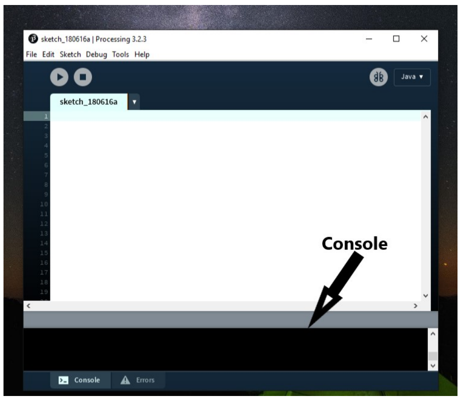
Exercise 2.2 Your program automatically saves the width and height of your window. Anywhere in your program if you call width or height, your program will refer to values of the width and height that you input in your size() function.
Setup a new window which is 400 pixels wide and 500 pixels high. Initiate a variable and assign it a value of 1. Call the rect() function with the first two parameters set as the variable name and set its width and height to half the window size. As your program runs, keep incrementing your variable by 10.
Your program should construct a series of rectangles drawn diagonally on your drawing window.
■

Exercise 2.3 Now initiate 2 more variables and assign them a value of 1. Give a color to your rectangle using fill(). Use your variables as parameters for fill() and keep incrementing your variables.
Your program should again construct a series of rectangles drawn diagonally on your drawing window except that every rectangle now has a different color. ■

2.2 Print Statements

2.2.1 Console

Console is the black area at the bottom of the processing window. This area can be used to display messages or any other information that a user intends to display. For example, it can be used to print simple statements or to keep a track of any particular program. enabling a programmer to understand what is happening inside a code.


sketch_180616a | Processing 3.2.3
File Edit Sketch Debug Tools Help
Java
sketch_180616a
Console
Console
Errors

The following the code, for example, prints the value of the background function as the value changes. This may enable a user to know the background color as it is displayed in the processing window.

■ **Example 2.8** Python Example Code

```
def setup():
    size(500,500)
    x = 0
def draw():
    global x
    background(x)
    x = x + 1
    print(x)
```

■

Any lines of code inside a draw() function keeps repeating. This program will run continuously unless you close your programming window or set up a condition to stop your program. For example, you may want to stop your program when the background color becomes red, or when the value of x exceeds 100. This could be achieved using conditionals which are covered next.

2.2.2 Syntax

`print()` and `println()` functions can be used to write text to the console. Anything contained within the parenthesis will be displayed at the bottom of the processing window.
The following code prints PROCESSING in the console area.

■ **Example 2.9** Python Example Code

```
def setup():
    size(500,500)
def draw():
    print("PROCESSING")
```

Output:
» PROCESSING ■

2.3 Conditionals

Conditionals are basically if-else statements that allow a user to run a specific code only if certain conditions are fulfilled. For example, you may assign a value to a variable and want your program to print the value only if it's even. Such a program will only print even values and ignore the odd values unless specified. The following code, for example, prints EVEN if the value is even and ODD if it is odd.

■ **Example 2.10** Python Example Code

```
def setup():
    size(500,500)
def draw():
    number = random(90)
    if number%2==0:
       print("EVEN")
    else:
       print("ODD")
```
■

The random() function generates random numbers. It takes the range of the numbers as parameters, for example, random(10,50) will generate a random number in between 10 and 50. The function will keep generating random numbers one at a time as long as your code keeps repeating. If you input only one parameter then your program will take it as the upper limit and the function will generate values between zero and the parameter you input.
Find more about `random`
`https://processing.org/reference/random_.html`

Exercise 2.4 Write a program that generates random numbers(**integers only** in the range (0,100) and prints if the number is divisible by 5 or not. For example, if the number generated is 26, then the console should display:

26 is not divisible by 5

■

2.4 Exercises

1. Use Multiplication to create a series of lines with increasing space between each line. *You'll have to use the line() function for this exercise.*
2. Draw a line on your processing window and using conditionals and variables make it move across the screen.
3. Modify the previous code such that when your line reaches one edge of your screen, it starts move in the opposite direction.

3. Loops

3.1 Introduction

The purpose of a loop is to keep repeating a set of lines of your code until a particular specified condition is reached. If your program does not specify any stopping condition, then your code will keep running infinitely until you close you program. In processing, any lines of code inside the **draw()** function are basically inside a loop, i.e, keeps repeating infinitely. The noLoop() function prevents your code from running continuously inside draw().
Two basic loops are the for and the while loop which will be covered in the following sections.

3.2 For loop

The for loop is used for lines of code that are to be implemented in a sequence. This loop can be used in two ways. It can either be used to iterate over your input in a specified range or can directly be used to access elements through the length of your input. Both of these methods are explained below.
The for loop that takes range as parameter will iterate over your input only in the specified range. Consider the following example code.

■ **Example 3.1** Python Example Code

```
for i in range(0,10):
        print(i)
```
■

This code will print numbers from 0 to 9 on your console. In the example above, i is any random alphabet that stores the range values as your loop proceeds. For example, at the first iteration i stores 0, in the second iteration i becomes 1 and so on. You could use any other alphabet and you would still have the same output.
Code that has to be repeated is placed inside the for loop and properly indented. For example, `print(i)` is placed inside the for loop because we want to print the value of i at every iteration. When you print i, your console prints the number of iteration your loop is on. The output is printed

on separate lines because it is printed as your program repeats. To avoid printing on separate lines you could use print(i,end=""). This syntax would print your entire output on the same line which in this case is : ***0 1 2 3 4 5 6 7 8 9***
If you want a comma after every number, you can use print(i, end=","). This is going to put a comma after every iteration.
If n is your higher range, your for loop will always run n-1 times, i.e., if you want to run your loop ten times, then your range should be from 0 to 11.
This type of for loop(range) actually takes 3 parameters. One is the lower limit, second is the higher limit and the third one is the step parameter. The step parameter tells your program how many numbers to skip to get to the next number. For example, if your for loop looks like:

```
for i in range(0,100,10):
```

then, the value of i begins at 0, steps 10 numbers and assigns i a value of 20. Your program will proceed with the value of i as *10,20,30,40* .... and terminate as i reaches 90.
You can make your for loop run in the reverse order too. For example,

```
for i in range(100,0,10):
```

will initiate your i with a value of 100 and terminate the program in 10 iterations as your i reaches 10.

Another way, in which the for loop is used, is by directly accessing elements instead of the count of the iteration. For example, if you wish to iterate over a string, the former method will not allow you to access the elements of the string directly since it works on numbers. To iterate over a string, you could use your loop in the following way:

■ **Example 3.2** Python Example Code

```
string = "Processing"
for i in string:
    print(i)
```

■

Here, your letter i is not the count of the iteration, but the letter which your loop is on.
At the first iteration the loop is on the letter "P", on the second iteration it is on the letter "r" and so on. Hence, this method is easier to directly access elements in a string. In the coming chapters, you will learn to use the range method to iterate over a string using string manipulation.
The output of this code would be the letters of "processing" printed on separate lines.

■ **Example 3.3** In programming, loops help you manage your code as they help to make your code precise and will do more work in lesser lines.
For example, the following code draws a number of ellipses creating a pattern.

```
def setup():
    size(500,500)
def draw():
    for i in range(79,0,-10):
        ellipse(250,250,i,i)
```

You can easily fill your screen with ellipses and draw a number of them by just adjusting the range of your loop. Without loops, you would have to re-write ellipse() with different parameters over and over again to get the same output.
With the code above, your first ellipse will have the parameters ellipse(250,250,79,79), in the second iteration the parameters will change to ellipse(250,250,69,69) and so on. Your program initiates the value of i with 79 which decrements to 0, skipping 10 values at every step.

Can you guess why we'd not be able to achieve our desired output if the loop was not made to run in reverse i.e. 79 to 0 instead of 0 to 79?
Try running the loop oppositely and see if you're able to get the same pattern. ■

Exercise 3.1 Use the code in example 3.3 and conditionals to create an animated hypnotizing pattern.
You will have to use the conditionals to alternately fill your ellipses black and white, and shorten the distance between every ellipse. Your draw function will give it the hypnotizing effect. ■

3.3 While loop

A while loop will keep repeating a set of instructions or code until a certain specified condition becomes either True or False. If your program does not specify the condition, your loop may keep running until the program crashes.

While loops and for loops can both be used to get the same output depending upon the user's preference. A while loop has three parts. In a for loop your i(any randomly chosen alphabet) starts from 0 (unless specified otherwise) and keeps incrementing until it reaches your required limit. In a while loop, you will have to initiate a variable giving it an initial value, then specify the limit or condition on which your loop has to stop running, and thirdly increase the value of i at every iteration. The first and the third tasks are automatically accomplished in a for loop, but you will have to specify those conditions in a while loop.

Our example codes from section 3.1 can easily be implemented using a while loop as follows:

■ **Example 3.4 Python Example Code**

```
i=0            #initiating a variable
while i<=10:   #Specifying the condition when it has to end
    print(i)
    i=i+1       #incrementing the variable at every iteration
```

■

This code prints exactly the same output as example 3.1.
Line # 1 initiates and assigns an initial value to a a variable. Line # 2 initiates your while loop giving it a condition to run as long as the value of i is less than or equal to 10 and line # 4 increments the value of i at every iteration.
Your program runs from top to bottom. When this code is executed, your program enters the while loop after assigning i a value of 0, and keeps repeating the while loop until i becomes 10. Any other line will not be executed until your program steps out of your loop. In the upcoming chapters you will learn to execute example 3.2 using while loop and string manipulation.

While loop works very similarly to its use in common English language. For example, consider the sentence "While your tea is not sweet, add half a spoon of sugar" according to which you are required to add half a spoon of sugar and taste your tea, if it is still not sweet you'd again add half a spoon of sugar and hence keep adding sugar until your tea is sweet enough for you. Your while loop also keeps running until your specified condition is fulfilled.
Here is another example.

■ **Example 3.5 Python Example Code**

```
number=1
while number<100:
    number = number * 2
    println(number)
```
■

The code prints the value of the variable number until the value is less than 100. At every iteration the value of number becomes 2 times the previous value and the value of number is printed after every iteration.

3.3.1 Using for and while loops interchangeably

A for loop is usually used when you know how many times your loop has to run beforehand, whereas a while loop is used if the number of iterations are not known or if the number of iterations the program is supposed to make is too high to keep count.

For example, if a programmer wishes to iterate over a string, he'd know that to able to iterate over every letter of the string the number of iterations required would be equal to the length of the string, but, for example, if a programmer wishes to add 2 to a variable and keep on adding 2 until the variable is less than 39, then he doesn't know the number of iterations his program will make until the variable becomes greater than 39. Those calculations are not required in a while loop as your loop will keep check of your variable and stop the program as soon as the variable becomes greater than 39.
It is true however, that everything that can be achieved through a for loop can also be achieved using a while loop and vice versa. Both of these loops can be interchangeably used. The following examples will help you understand the difference between the two and the need to use a more appropriate loop.

■ **Example 3.6** Python Example Code

- **for loop**

```
for i in range(100,0,-10):
    print(i)
```

- **while loop**

```
i=100
while i>=0:
    print(i)
    i = i - 10
```

This code prints numbers in the reverse order, i.e, from 100 to 0 stepping 10 numbers at every iteration. As you can see, it is more appropriate to use a for loop in this case. ■

■ **Example 3.7** Python example code

- **while loop**

```
i=1
factorial=1
while i<=number:
   factorial=factorial*i
print(factorial)
```

- **for loop**

```
factorial=1
for i in range(0,number):
    factorial = factorial*i
print(factorial)
```

This code calculates the factorial of any number that you enter in the number variable. You can use any of the two loops for this code. ■

Exercise 3.2 You probably have clear concepts regarding loops now. Now using loops fill your entire processing window with the polka dot pattern. Also give colors to your window on the dot. *You may use the point() function or the ellipse() function to get the desired output.* ■

Exercise 3.3 Using loops and conditionals fill your entire screen first with slant lines, all equally spaced and equally tilted. Secondly fill your entire screen with lines this time oppositely tilted from the previous one. This should create a diamond shaped texture.
Refer to the following link to get an idea about the pattern.
`https://www.pinterest.co.uk/pin/184506915955252302/` ■

3.4 Nested Loops

Placing one loop inside another loop is called nesting and the total number of iterations your loop makes is equal to the number of iterations of the outer loop multiplied by the number of iterations of the inner loop. The inner loop runs through a complete cycle for each single iteration of the outer loop.

■ **Example 3.8** Python Example Code

```
for i in range(0,5):
   for j in range(0,10):
      print(i,j)
```

The code above has 2 nested for loops. Once the code starts running, it first enters the first for loop and then enters the second for loop. The program keeps repeating the second for loop until the entire range is covered and then repeats the first for loop again. This loop again enters the second for loop and keeps repeating it until the entire range is covered before the third iteration of the first for loop.
Run the code on your system and see what it prints on your console. ■

Look at the following processing example using nested loops.

■ **Example 3.9** Python Example Code

```
for i in range(10,100):
    for j in range(10,100)
       point(x,y)
```
■

Here is another example followed by the output:

■ **Example 3.10** Python Example Code

```
noStroke()
for i in range(0, 100, 10)
   for j in range(0, 100, 10):
     fill((i+j) 8 1.4)
     rect(i, j, 10, 10, 10)
```
■

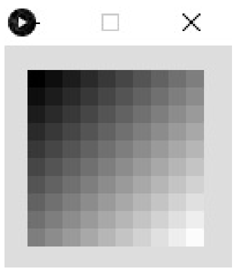


Figure 3.1: Example Sketch

Try replacing rect() with ellipse()

Exercise 3.4 Modify the code in 3.10 by using the line function instead of rect() such that it creates a smiliar visual pattern. ■

Exercise 3.5 Using **nested** loops fill your entire processing window with the polka dot pattern. Also give colors to your window and the dots. *You may use the point() function or the ellipse() function to get the desired output.* ■

3.5 Python break and continue statements

Loops require some condition to allow a programmer to step out of the loop and execute other lines of code, but in some cases a programmer may temporarily or permanently break out of a loop before the specific condition is reached. For this purpose, the break and continue statements are used to alter the normal flow of a loop.

3.5.1 Python break statement

The break function is used to immediately let a programmer break through a loop and run the next lines of a code. This statement breaks only the current loop, for example, in case of nested loops the break statement only breaks the innermost loop while the outer one keeps running smoothly unless you have another break statement indented with it.

■ **Example 3.11** Python Example Code

```
var="Processing"
for i in var:
     if i=="s":
         break
```

This loop will run till it reaches the letter s and then the program breaks out of it. ■

3.5.2 The continue statement

The continue statement allows a user to restart a loop without executing the code completely. The code doesn't start from the very first iteration rather it skips the lines of code that are placed after the continue statement and goes on to the next iteration.

■ **Example 3.12** Python Example Code

```
var="processing"
for i in var:
    if i=="s":
        print(i)
        continue
```

This code will skip the print function when the program reaches the letter s and continue with the rest of the iteration. ■

4. Functions

4.1 Introduction

We have been using the term 'function' for some time now. For example, to draw an ellipse or a circle we've been using the ellipse() **function** and to draw a line we've considered the line() **function**. So far, we know that a function in programming is any command that you input and give it it's required parameters according to the output you want. Every function performs its specific task, for example, a line() function draws a line on your window and an ellipse() function draws an ellipse on your window. Both of these however, are built in functions, i.e., functions that your software provides you. You do not need to write the code for your program to draw a line because your software already provides you that function. You are, however, required to write code for more complicated tasks. In processing, for example, the setup() function which sets the size of your window is a function that you define to set the size of your display. Similarly, the draw() function repeats any lines of code that you input inside it. If you write background(90) in your draw function, your function would simply set the background of your window unless you write more lines of code for it to execute.

4.2 Built in functions

The **draw()** and the **setup()** functions are functions that you initiate on your window and they execute any lines of code that you write. The line() and the ellipse() functions do not work the same way. All you do, is call these functions and they draw a line or an ellipse according to the parameters that you input. The lines of code executed by these functions are not visible to the programmer because they are already fed into your program. These are built in functions. These functions also execute some lines of code to draw a line or an ellipse on your window exactly like the draw() function, but those lines of code are already defined in your program and you do not need to write them over and over again. Unlike the draw() you do not need to initiate these functions and write their code. Using these built in functions, you can work efficiently, save time and make more complicated functions. For example, using the line() function with different parameters you can draw a rectangle and perform complicated animations in processing. rect() is also a built in for rectangle.

Here is a simple code where draw() constructs a line at the cursor position and keeps constructing the line as long as your code runs.

■ **Example 4.1** Python Example Code

```
def setup():
    size(500, 500)
def draw():
    line(0, 0, mouseX, mouseY)
```
■

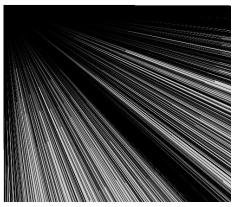


Figure 4.1: Example Sketch

4.3 Parameters

Built in functions usually take parameters as inputs, for example, line() takes four parameters which define its position. The first two parameters are the x and y coordinates of the starting point of the line, whereas the other two are the x and y coordinates of the end point of your line. Similarly, ellipse() function takes 4 parameters, two of which define position of the center of ellipse and the other two are the measures of it's height and width in pixels.

On the other hand,Your draw() and setup() functions take no parameters, therefore the parenthesis placed after their name are empty parenthesis. You do not need to call these functions too, unlike line() or ellipse(). These functions are called by themselves in processing. All of the other functions that you define need to be called. You will further understand the syntax in the following section.

4.4 Syntax

In Python, **def** is the keyword for defining functions. If you intend to initiate a function of your own, you would need to name that function and give it its parameters(if it takes any). The syntax is def followed by the name of your function, further followed by parenthesis and a colon as shown below.

```
def functionName():
```

The function named `functionName` here takes no parameters.

```
def newFunction(x1,x2):
```

The newFunction() here takes 2 parameters, x1 and x2.

Consider the following example:

■ **Example 4.2** Python Example Code

```
def setup():
    size(100, 100)
    noStroke()
def draw():
    ellipse(50, 50, 60, 60)
    ellipse(50+10, 50, 30, 30)
    ellipse(50+16, 45, 6, 6)
```
■

This example draws an eye on your window, but we know that the draw() function is usually used for lines of code that need to repeat themselves to create animation, so wouldn't it be better if we have another function for drawing an eye:

■ **Example 4.3** Python Example Code

```
def setup():
    size(100, 100)
    noStroke()
def draw():
    background(204)
    eye(65, 44)
def eye(x, y):
    fill(254)
    ellipse(x, y, 60, 60)
    fill(0)
    ellipse(x+10, y, 30, 30)
    fill(255)
    ellipse(x+16, y-5, 6, 6)
```
Output:

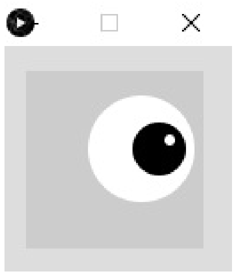


Figure 4.2: Example Sketch

■

Exercise 4.1 Modify example 4.3 such that it draw a pair of eyes rather than a single eye. *hint: you just need to call the eye function twice using different parameters* ■

4.5 Transformations

4.5.1 Translations

The processing window allows you to adjust the coordinate system so as to be able to move and resize shapes without changing their parameters. For example, if you want your program to draw 2 rectangles with respect to each others position at two different positions on your processing window, you may use translation of the coordinate system. In this way you can draw your rectangle without changing it's parameters. Translations allow you to adjust your processing window by shifting it's origin.
The syntax for translate is `translate(x,y)`.

The follwing example is how translation works.

■ **Example 4.4** Python Example Code

```
def setup():
    size(500,500)
def draw():
    rect(250,250,100,100)
    translate(20,50)
    rect(250,250,100,100)
```

■

The translate function is not very useful in this case, because you already have to rewrite the rect() function again. But you can easily manage your code by calling the rect() function repeatedly in a loop to draw the same shape again and again at different positions on your window. The following code will help you understand.

■ **Example 4.5** Python Example Code

```
def setup():
   size(500,500)
def draw():
   for i in range(0,20):
       translate(20,10)
       rect(0, 0, 100, 100)
```

■

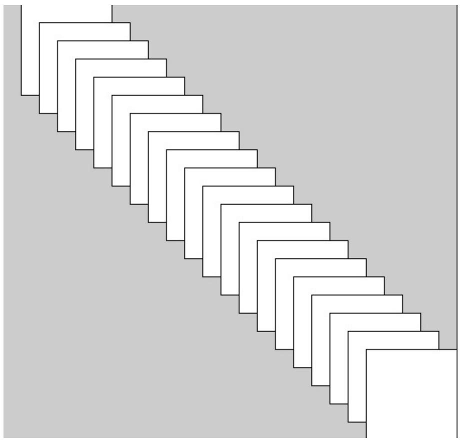


Figure 4.3: Example Sketch

In it's default state, the origin is located at the top-right of your window. In the example above, you loop draws a series of 20 rectangles each of them translated (20,10) units from your previous coordinate system. This is because the translate() function is additive. Unless your draw function repeats, the translate function keeps your coordinate system in the same coordinates that you set in your code at every iteration.

For example, if one line of your code is `translate(20,20)` and any other line in the same code is `translate(40,30)`, then the total transformation will sum up to `translate(60,50)` for that specific code. When the draw() function repeats your code will set your coordinate system to it's default state.

Exercise 4.2 Look up the pushMatrix() and popMatrix() function and understand their working. Create a program and define a function named rec() which draws a series of 5 rectangles 100 units high and 150 units wide whenever the mouse is pressed. *hint: Use loops and transformations.* ■

4.5.2 Rotations

Translations allowed you to adjust the coordinate system such that the origin is moved and so are your shapes easily placed at different locations on your window without changing their parameters. The rotate() function allows you to rotate the coordinate system, so that your shapes may be tilted or rotated on your screen. The parameter to rotate is the angle which sets the amount of rotation of your shapes. Your shapes are always rotated in the clockwise direction if your angle is positive. Negative angles rotate your function in the anti-clockwise direction. Like translate(), the rotate() function is additive. If anywhere in your program you set the rotation to $\frac{PI}{4}$ and then at any other line you set it to $\frac{PI}{4}$ again, any shape drawn after this will be rotated $\frac{PI}{2}$ radians unless the draw() function resets the coordinate system to its default state.
Try out the following examples to clearly understand rotations.

■ **Example 4.6** Python Example Code

```
rect(55,0,30,45)
rotate(PI/8) #clockwise rotation
rect(55,0,30,45)
```
■

■ **Example 4.7** Python Example Code
```
translate(45,60)
rect(-35, -5, 70,10)
rotate(-PI/8)
rect(-35, -5, 70, 10)
rotate(-PI/8)
rect(-35,-5,70,10)
```
■

4.5.3 Scale

Scale helps you to rescale your shapes. It allows you to control the size of the shapes. It takes either 1 or 2 parameters. With 2 parameters, it scales the x and y axis separately. With a single parameter it scales the entire shape equally. The following 2 examples illustrate how a single parameter works differently from 2 parameters.

■ **Example 4.8** Python Example Code
```
ellipse(32,32,30,30)
scale(1.8) # scales the shape to 180%
ellipse(32,32,30,30)
```

■

■ **Example 4.9** Python Example Code

```
ellipse(32,32,30,30)
scale(2.8,1.8) # scales the x-axis to 280% and the y-axis to 180%
ellipse(32,32,30,30)
```

■

Exercise 4.3 You can use the additive property of rotations and translations to continuously rotate a line and create a circle. The example code in 4.7 is one way of using the rect() function to create a circle.
Your task is to use the **line()** function to create a color wheel such that the colors merge into each other. Your wheel may not include all the colors. ■

5. Functions(2): Return Statements and Recursion

5.1 Return values and function statements

You've been using the processing software to draw shapes and figures and to print values. These tasks are only direct commands that you ask your program to do. In most cases, your program is expected to return an output in a specific data type, for example, you may write a program to calculate the product of two numbers and return you the output. The output may be float or an integer. Your return statement will immediately exit the function without executing any other lines of code.
Functions that do not return any value, for example, functions that are only required to print on the console or draw some shapes are included in the special type **void()**.
void() is specially used to declare functions that do not return any value. For all the other functions, you have to declare the function with the data type it is expected to return.
The Python programming language allows you to create functions without specifying their data type, so void() is not used to declare functions in Python. Those functions that do not return any value, return None and are still categorized as void() in the Python.
The following examples make calculations and return the desired values. The setup and the draw function do not return any value and are therefore always the void data type.

■ **Example 5.1** Python Example Code

```
def setup():
    float c = fahrenheitToCelcius(451.0)
    println(c)
float fahrenheitToCelcius(float t):
    float f = (t-32.0) * (5.0/9.0)
    return f
```
■

5.2 Default Arguments

When you define your own functions, it is your choice to either define the function with some parameters or no parameters at all. If while defining your own functions you give it some parameters, then you are also required to give it those parameters whenever you call that function, otherwise your program will return an error. These parameters are usually variables and you may assign them any value with which your program will run.
Another way arguments work is by assigning them default values. Default arguments are also passed similar to variables and the only difference is that you do not need to pass default values while making a function call as these values are already assigned and set to default. However, it is possible to pass a value to a default argument and in this case that value over rides the default value.

■ **Example 5.2** Python Example Code

```
def Calculate(a, b , operation = True)
    while operation==True:
        if a>b:
            return a-b
Calculate(10,3)
```

Output :

»7

```
Calculate(10,3,False)
```

Output :

» None

5.3 Composition

Programming allows you to call a single function multiple times or a function within a function. For example, consider the following example code:

■ **Example 5.3** Python Example Code

```
def Calculate(a, b):
    val = a+b
    return val
x = Calculate ( Calculate(3,4), Calculate(10,10))
print(Calculate(x, Calculate(20,10)
```

Output:

»57 ■

. Here is how it works:
Your function is required to calculate the sum of any two numbers that you pass as arguments in the function call.
Your first function call is made in the variable x which works as follows.

```
x = CalculateCalculate(3,4), Calculate(10,10))
x = Calculate(7,20)
x = 27
Next function call on line # 5 works as follows:
print(Calculate(x, Calculate(20,10))
print(Calculate(27, 30))
print(57)
```

Output:

»57 ■

Exercise 5.1 Write a function to convert inches into centimeter and return the calculated value. ■

5.4 Recursion

In the previous few chapters, you were introduced to loops and how they enables a programmer to repeat a set of lines of code. You also learned to define functions of your own and call them whenever you want to execute the respective lines. It is also possible to call a function inside a loop if you want to repeat it over and over again.
In this section we will look at another way of repeating a set of lines of code called **Recursion**. To understand the concept of recursion, a very common example is to stand between two mirrors and see infinite reflections of yourself.
Recursion allows a function call to itself again and again until a specified condition is reached. This condition is called the base case and your program returns you your output when you reach the base case of your program. If you do not specify the stopping condition, the function will keep calling itself infinitely. Recursion does the same work as loops; i.e., repeats a set of lines of code until the stopping condition is reached. It is important to understand the concept because recursion makes it easier to understand the working of a code.

■ **Example 5.4** Python Example Code

```
def rec_exp():
    print("Processing")
rec_exp()
```

Output:

```
»Processing
```
■

■ **Example 5.5** Python Example Code

```
def rec_exp():
    print("Processing")
    rec_exp() #Function call inside a function
rec_exp()
```

Output:

```
»Processing
»Processing
»Processing
»Processing
```
and so on. ■

This code will keep repeating as the function call has been made inside the function and no stopping condition or the base case has been defined. This is thus, a recursive function running infinitely. To prevent your function from running infinitely you are required to write a base case. for example, you may want this function to run 10 times only, then your function may look something like:

■ **Example 5.6** Python Example Code

```
def rec_exp(count):
     if count == 0:
         return
     else:
         print("Processing)
         rec_exp(count-1)
rec_exp(10)
```

■

Here, the if condition is your base case. When this condition is fulfilled, the program stops running. Otherwise your program keeps repeating the same function with the value of count decrementing by 1 every time the function is called.

Another example is a program to calculate the factorial of a number.

■ **Example 5.7** Python Example Code

```
def fact(n):
   if n < = 1:  base case
         return 1
    else:
         return n*fact(n-1)
```

■

Line # 9 of this function calls the function again and again until the base case is reached
Can you write the same program using loops?

The following example uses recursion to draw a series of lines by calling `drawLines` over and over again.

■ **Example 5.8** Python Example Code

```
 def setup():
     size(100, 100)
    drawLines(5, 15)
def drawLines(x, num):
     line(x, 20, x, 80)
     if num > 0:
         drawLines(x + 5, num -1)
```

■

Output:

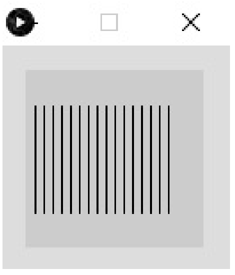


Figure 5.1: Example Sketch

Exercise 5.2 Write function to draw a shape. Use it to draw the shape to the screen multiple times, each at a different position. ■

Exercise 5.3 Rewrite example 8.4 using loops to produce the exactly same output. ■

6. Strings

We already had a short introduction to strings when we were studying about different data types. The `char` data type stores a single letter, symbol or digit. Variable assignment for the `char` data type works exactly similarly to `int` and `float`. The string data type stores sentences and words. Any text enclosed in single or double quotes in python creates a string and the `char` data type is basically a string in Python.
String is different from `char, int and float` data types because string is a **class** and therefore you can perform multiple operations on strings using string **methods**. Variables are declared in a similar way as in the other data types as

```
a = "Processing"
print(a) # prints Processing
```

Here, the variable `a` is not just a variable, but an object. All variables created with the `String data type` are objects, i.e., you can perform some operations on those variables. For example, every String object has a method to capitalize it's letters. Other methods include extracting parts of strings and comparing two strings etc, all of which are introduced later in this chapter.
The dot operator is used to access methods for any object as shown in the example below.

6.0.1 String Methods

length

len returns the length of the string, i.e., the number of characters in a string.

■ **Example 6.1** Python Example Code

```
string1 = "Processing"
string2 = "hi"
print(len(string1))
Output:
»10
print(len(string2))
Output:
»2
```
■

lower()

lower() returns your string with all of its letters converted to lower case.

■ **Example 6.2** Python Example Code

```
string1="PROCESSING"
print(string1.lower())

Output:
»processing
```
■

For any variable e.g. a, a.islower() will return True if all of the letters of the string are in the lowercase and False in the other case.

upper()

upper() returns your string with all of its letters converted to uppercase.

■ **Example 6.3** Python Example Code

```
string1="processing"
print(string1.upper())
```
Output:
»PROCESSSING
■

For any variable, variable.isupper() will return True if all letters of your string are in the uppercase and False if vice versa.

strip()

The **strip()** method returns your string with all the whitespaces at the beginning and the end removed.

■ **Example 6.4** Python Example Code

```
var = "       Processing "
print(var.strip())
```
Output:
» Processing ■

split()

The **split()** method takes a delimiter and returns a list of your string separated into substrings by the given delimiter.

■ **Example 6.5** Python Example Code

```
var = "Hello world"
print(v.split())
```

Output: ["Hello", "world"] ■

■ **Example 6.6** Python Example Code

```
var= "hello-world"
print(var.split("-"))
```

Output:

» ["hello", "world"] ■

ord() is a built-in in Python that is used to find the integer value of a character according to ASCII codes. For e.g., "a" has an integer value 97 whereas "A" has an integer value is 65.

```
»print(ord("A"))
»65
```

chr is also a built-in which works exactly oppositely to ord(). chr() takes an integer value and returns the respective character.

```
»print(chr(65))
»"A"
```

Read more about chr() and ord() at https://docs.python.org/3/library/functions.html#ord

Exercise 6.1 Write a function that takes a string and returns a new string with all the white spaces and special characters removed. *hint: Use ASCII codes to check for the special characters.* ■

6.1 String Slicing and Indexing

In Python, it is possible to access the elements of a string. Square brackets are used to access individual elements or to slice along the index to obtain a substring.

■ **Example 6.7** Python Example Code

```
string1 = "Hello"
print(string1[1]) #prints the letter at position 1
```

Output :

» e ■

It is also possible to access more than 1 character, i.e., a sequence of characters as:

```
string[start:end]
```

■ **Example 6.8** Python Example Code

```
print(string1[1:3])
Output:
»el
```
■

In any case if you do not know the ending index of your string, you may use:

```
string[start:]
```

where your program will access the elements from the start through the rest of your string or you may use `string[:]` to access the entire string from start to end.

You can also a third parameter i.e. step to skip elements from your string while slicing: The following syntax will print alternate elements.

`string[start:end:2]` or `string[::2]`

Negative Indexing

In python, you can use negative indexing to directly access elements at the end of your string. For example, the index -1 refers to the last element of your string whereas -2 refers to the second last element of your string and so on. In this way, you can use negative indexing to access your entire string.

■ **Example 6.9** Python Example Code

```
string1 = "RELAX"
print(string1[-1])
```

Output:
» X ■

6.2 Updating Strings

Strings are immutable, i.e., once you assign a string to a variable you cannot make any changes to it, however, it is possible to update an existing variable.

■ **Example 6.10** Python Example Code

```
var = "Hello"
print(var + "World")
Output:  HelloWorld
```
■

■ **Example 6.11** Python Example Code

```
  var = "Hello"
var = var + " World"
print(var)
```

Output:
» Hello World ■

In example 6.11, line # 2 updates the variable *var* to Hello World.

6.3 String operators

The following table lists some of the common string operators:

Operators	Operation	Example
+	Concatenation- Concatenates 2 strings	"Hello" + "world" » Helloworld
*	Repetition	"Hello" * 3 » HelloHelloHello
[]	Accessing elements are the given index	"hello"[1]» h
[:]	Range Slicing	"hello"[0:3] » hel

Exercise 6.2 Write a program to check if a given string is a palindrome. A palindrome is a word that spells the same way backwards as forwards. (All white spaces and special characters are ignored). For example, LOL is a palindrome and so is "MADAM! I AM ADAM". ■

Exercise 6.3 Two strings are said to be anagrams if they both have the exactly same letters, for example, MARY and ARMY are anagrams. Write a program that checks if two given strings are anagrams. ■

7. Strings(2): Files

7.1 Introduction

When you run a program your system stores certain amount of data, but as soon as your program stops the entire data is lost, since it had only been temporarily stored on your system only as long as your program is running. You also cannot access the data anywhere outside the program. Hence, to be able to access data even after our program stops running or to store useful information that your program may be generating, it is possible to store your data in files or use data from other files to start your program.
All software files have a specific format to interpret data as it is read from the memory and this format is often referred to by the extensions of the files. TXT for example, is the extension for plain text files and MP3 is used for storing sound.

7.2 Export files

`PrintWriter` is a class that allows you to export files. One of it's methods `createWriter()` opens a file to write to and add data to the file as long as your program is running. `flush()` method is used to let your program save your file correctly and the `close()` method finishes writing the file. Consider the example code on the next page:

■ **Example 7.1** Python Example Code

```
def setup():
    size(500,500)
    output = createWriter("newfile.txt")
count = 0
def draw():
    global count
    count = count + 1
    output.println(count)
def keyPressed():
    output.flush()
    output.close()
    exit()
```
■

Here is how it works:
Line # 3 initiates a file named "newfile" and assigns it to a variable `output`.
Line # 8 writes the value of count to the file and the declared function keyPressed() stops the code as soon as any key is pressed and file writing ends.

Exercise 7.1 Write a program that takes the area of a square as input, calculates the dimensions and writes the dimensions along with the area on your file. Your setup window should take one of the dimension as input for the fill() function to fill your respective rectangle and draw a series of rectangles starting at the top left corner of your window with the dimensions reducing to zero decrementing by 3 each time. Your program will stop running when the dimensions become zero. ■

7.3 File Reading

It is possible to load external data into your pragram by the required data from some external file. A plain txt file enables you to carry out this operation as there is no formatting or colors in the file. The built in function `loadstrings()` loads the data of the file by reading the contents of a file and creating an array of its individual lines. Each line of a file is a single element in an **array**. You will learn more about arrays in the coming chapters.
Every file that you create is created in the sketch folder. Similarly, any file that you want your program to read must be located in the sketch folder.
The syntax for file reading is as follows:

```
String[] lines = loadStrings("numbers.txt")
```

Here the `lines` array is declared and assigned to `String`. Every line is one element in the array and your code will read through each line and print it to the console.

To print the entire files to the console you will have to print all the lines in the file. At one time it is possible to print one line as every element of your array is one entire line of your file. For example:
`Processing is very interesting`
`We're about to finish learning` it will be stored in your array as:

array = ["Processing is so interesting", "We're about to finish learning it"]
Accessing an element of an array works exactly similarly to accessing an element of a string, i.e.,

array[index] will give you the element at the specific index.

Here array[0] will return "Processing is so interesting".
The following code thus prints the entire file,

■ **Example 7.2** Python Example Code

```
for i in range(0, len(lines)):
    println(lines[i]) #where your program prints the element at the index as i increases.
```
■

7.4 Tables

7.4.1 Creating Tables

So far we learned how to load and read a plain text file, but what if you want your program to read a plain text file and load it in a different way. For example, you may want your program to load plain data in a table format to understand it more clearly!
The `Table` class in processing allows you to load data in the table format. It may be useful to manage data or combine multiple types of data.
Creating a simple table is pretty simple in processing. Like every other class, you need to declare the class outside your function and generate a new table inside the setup() function. You can construct your table using methods such as addColumn() and addRow() and save your file at the end of your program. Since you're constructing a table you will have to save it as a csv file.

This code constructs a simple table.

■ **Example 7.3** Python Example Code

```
def setup():
    table = new Table()
    table.addColumn("id")
    table.addColumn("species")
    table.addColumn("name")
    saveTable(table, "data/new.csv")
```
■

Go to the sketch folder and open the file that you just created. You will see 3 columns on your excel window.

7.4.2 Loading Tables

As in plain text files, it is also possible to read tables. To be able to load a table and print it's contents, the built in function loadTable() is used. The table you load on your program must be located in your Sketch folder. For this, go to the Sketch option on your processing window and select *Add File* to add your respective file.

Consider the following code:

■ **Example 7.4** Python Example Code

```
def setup():
    table = loadTable("new.csv", "header")
    println(str(table.getRowCount()) + "total rows in table")
    for row in table.rows():
        id = row.getInt("id")
        species = row.getString("species")
        name = row.getString("name")
        println(str(name) + "(" + str(species) + ") has an id of " +str(id))
```

The following link will help you understand the code.
`https://processing.org/reference/loadTable_.html` ■

This code reads a file named *new* and prints its contents. Line #2 loads the file, so if your program returns an error on this line then your file is probably not in the correct folder. The next line prints the number of rows in the table using the method `table.getRowCount()`. The for loop iterates over the **header** and allows you to extract data using the titles of individual columns. The header is basically used to tell your program that the first row in your file categorizes every column of your data file and hence you can refer to individual columns using the categories in the header. You can search the data in your tables using table methods such as findRow() and matchRow().

Exercise 7.2 Look up different Table methods at `https://processing.org/reference/Table.html` and attempt the task given.
Construct the table shown in section 6.3 and the read the contents of the table by printing them to your console. ■

8. Lists

Lists are one of the most frequently used **collection data types** or **sequences** in Python. This particular data type is an ordered collection data type that allows you to make changes to it using a number of methods. Lists therefore, unlike strings are mutable.
Any data enclosed in square brackets creates a list. For example, the following code creates a list and assigns it to the variable `var`.

■ **Example 8.1** Python Example Code

```
var = ["apple", "banana", "mango"]
print(var)
```
■

List is also a built in function in python. It takes a string and creates a list of it's elements. The .split() method of strings also creates a list of a string by splitting it. Both of these methods work differently.
The following code will help you understand.

■ **Example 8.2** Python Example Code

```
var = "Python"
print(list(var))
```

Output:

```
» ["P", "y", "t", "h", "o", "n"]
```

```
print(var.split())
```

Output:

```
» ["Python"]
```
■

The .split() method will return your string as the only element of the list if there are no blank spaces in your string or if you do not pass any delimiter, while the list method will split the entire string such that every letter is a single element of your list.

Lists are mutable. Thus, it allows you to make changes to your lists unlike strings where once you assign it to a variable, all you can do is make a copy with any alterations you would want to make but cannot change the original string. In lists you can make changes to the original variable.

■ **Example 8.3** Python Example Code
var = ["apple", "banana", "mango"] var[1] = "cherry" print(var)

Output

» ["apple","cherry", "mango"] ■

In the code above, line #2 accesses the second element of the list and sets it equal to cherry. When you output your list it returns you a list with the respective changes made.

8.1 Creating Lists

There are several ways of creating or initializing a list. One way to do so is using the built in method `list`. The following method creates an empty list.

■ **Example 8.4** Python Example Code
```
 newlist= list()
newlist1= list(("apple", "banana", "mango"))
```
■

In this code, `newlist` creates an empty list, whereas `newlist1` creates a list of fruits.
For this method used in example 8.4, if you pass a string, it will split the entire string and create a list with all of the elements of your string as individual elements of the list.
If you pass another list, it will create a copy. and if you pass a tuple i.e list of elements enclosed in parenthesis then this method will convert it into a list of those elements.
Another way of creating a list is:

■ **Example 8.5** Python Example Code
```
newlist=[]  #Creates an empty list
newlist1=["apple","banana", "mango"] # Creates a list of fruits
```
■

8.2 List slicing

List slicing is not very different from string slicing as you can easily access a sublist or some part of your list from the original list. The syntax is using square brackets containing the starting and ending indexes of your desired sublist placed immediately after your original list as shown below.

■ **Example 8.6** Python Example Code
```
list1 = ["apple","banana","mango","cherry"]
list1[2:4]
```
Output:

» ["mango","cherry"] ■

8.3 Modifying Lists

List slicing plays a major role in modifying your lists along with their mutability that allows you to make changes to them. You can slice any list and store the sliced part in a variable and replace elements of your list with the respective slices. List Modification is further supported by a number of list methods that allow you to add, remove and replace elements. These methods are extremely important as they allow you to work with lists more effectively. The next section provides a brief description of some of these methods.

8.4 List methods

append()

append() allows you to add any element at the end of your list. For example, the following code creates a list and appends an element at the end.

■ **Example 8.7** Python Example Code

```
newlist=list(("apple", "banana", "mango"))
newlist.append("strawberry")
print(newlist)
```

Output:

```
» ["apple", "banana", "mango", "strawberry"]
```

■

remove()

remove() allows you to remove any element from your list.

■ **Example 8.8** Python Example Code

```
newlist= ["apple", "banana", "mango"]
newlist.remove("banana")
print(newlist)
```

Output:

```
» ["apple","mango"]
```

■

pop()

pop() takes an index and removes an element from the list on that index and allows you to store the removed element in a variable. For example:

■ **Example 8.9** Python Example Code

```
newlist= ["apple", "banana", "mango"]
var= newlist.pop(1)
print(newlist)
print(var)
```

Output:

```
» ["apple", "mango"]
»banana
```

■

join()

This method allows you to convert your list into a string by joining elements of your list. You may join your list using a dot or a comma which is specified while using the built in.

■ **Example 8.10** Python Example Code

```
newlist1= ["Coding","is","a","mess"]
print("-".join(newlist1))
```

Output:

```
» "Coding-is-a-mess"
```

■

There are a number of other list methods which include:

sort()
- sorts the list

insert()
- inserts an element at the specified index.

reverse()
- reverses your entire list

copy()
- creates a copy of your list.

You can read more at `https://www.programiz.com/python-programming/methods/list`

8.5 List Operations

Len

The **len** function returns the length of your list, i.e.,the total number of elements.

■ **Example 8.11** Python Example Code

```
list1= [1,2,3,4,5]
print(len(list1))
```

Output:

```
» 5
```
■

Concatenation

Concatenation allows you to combine two or more lists.

■ **Example 8.12** Python Example Code

```
list1 = [1,2,3]
list2 = [4,5,6]
print(list1+list2)
```

Output:

```
» [1,2,3,4,5,6]
```
■

Repitition

Multiplying your list by a number will repeat it's elements that number of times.

■ **Example 8.13** Python Example Code

```
list1=["processing"]
print(list1*4)
```

Output:

```
» ["processing", "processing", "processing", "processing"]
```
■

Exercise 8.1 Using loops generate a list containing numbers in the range (0,500). Split the odd and even numbers into two different lists. ■

Exercise 8.2 Write a function that iterates values in the list generated in the previous question in the reverse order and takes (value + 10) as the parameter for the height and width of your ellipse for even iterations and (value + 20) for odd number of iterations. Your pattern should look like figure shown below. ■

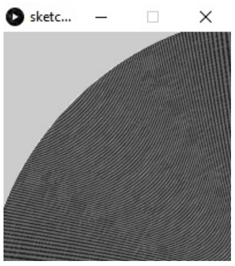


Figure 8.1: Example Sketch

Exercise 8.3 Create an empty list named `list_areas`. Write a function that uses random() to generate areas of equilateral triangles and calculates the length of each side. Append the lengths to list_areas and keep drawing the triangles on your window.
Area of an equilateral triangle = $\frac{\sqrt{3}}{4}(side)^2$

9. Lists(2)

9.1 The in operator

Python **in** operator allows you to look for any element in your list. For example, you may want to know if a particular element exists in your list, then you may use the **in** operator as:

■ **Example 9.1** Python Example Code

```
myList = [1,2,3,4,5,6,7,8,9]
if 1 in myList:
    return True
else:
    return False
```
■

9.2 Iterating Lists

Lists can be iterated like strings where loops allow you to iterate element by element over your list.

For loop

The **for-in** loop is the most easiest way to iterate over your list and get access to its elements one by one. The syntax for which is:

```
for element in list:
    print(element)
```

You may also use the range function and access the elements of your list using indexes as in strings, but one of the list methods is **enumerate()** that accesses both the index and the item directly from your list. Here is how it works:

■ **Example 9.2** Python Example Code
```
for index,element in enumerate(list):
    print(index,item)
```
■

This code will give you all the elements of the list along with their indexes. You may change the format of the output in the print statement.
Using only the **range** function will give you the index, so in any case where you do not need the elements of your list you may use the for loop as:

■ **Example 9.3** Python Example Code
```
for index in range(len(list)):
    print(index)
```
■

The next example code stores data into a list and iterates the list to use the data to draw a series of rectangles.

■ **Example 9.4** Python Example Code
```
x = [ 50, 61, 83, 69, 71, 50, 29, 31, 17, 39]
fill(0)
for i in range(0, len(x))
    rect(0 , i * 10, x[i], 8)
```
■

Iterator Protocol

You can also iterate over your list without using loops. Python iterator protocol uses the __next__() function to to access elements of your lists one by one. Consider the following example:

■ **Example 9.5** Python Example Code
```
myList = [1,2,3,4,5,6,7,8,9]
x = iter(myList)
print(myList.__next__())
print(myList.__next__())
```
Output:
```
» 1, 2
```
■

This code will print the first two elements of your list. For printing more elements keep using __next__() function until you entire list is covered.

while loop

While loop can also be used for iteration in the same way.

■ **Example 9.6** Python Example Code
```
myList = ["apple", "banana", "mango","cherry"]
index = 0
while index < len(myList):
    print(myList[index])
    index = index + 1
```
■

To directly access the index, you can print index instead of myList[index]. You can also get both the element and the index by using `print(index, myList[index])`.

Here is another processing code, that draws a series of lines on your window using the while loop and line().

■ **Example 9.7** Python Example Code

```
data = [19, 40, 75, 76, 90]
count = 0
while count <= len(data):
    line( data[count], 0, data[count], 100)
```
■

9.3 List Comprehension

List comprehension is a way to create new list from your existing lists. The syntax for list comprehension is a statement followed by a for loop contained within square brackets. You can accomplish the same tasks using for/while loop but list comprehensions consumes less space making your code precise.
Here are two codes that do exactly the same work using list comprehension and for loop.

■ **Example 9.8** Python Example Code (List comprehension)

```
myList = [x*2 for x in range(10)]
print(myList)
Output:
» [0,1,2,6,8,10,12,14,16,18,20]
```
■

■ **Example 9.9** Python Example Code (for loop)

```
myList = []
for i in range(0,10):
    myList.append(i*2)
```
Output:
» [0,1,2,6,8,10,12,14,16,18,20] ■

9.4 Matrices and Nested Loops - Lists within lists

Accessing individual elements in a list is very simple, as a single loop can get those elements for you. Consider myList as the list of fruits from which you are required to access every fruit. You can do so using a single loop as follows:

■ **Example 9.10** Python Example Code

```
 myList = ["apple", "banana","cherry"]
for fruit in myList:
    print(fruit)
```
■

Now consider that every fruit in myList is itself a list, i.e. lists within a list. Your list would then look like:

```
myList = [["apple"], ["banana"],["cherry"]]
```
Now, the code in example 9.10 will give you a different output i.e. this time it will give you a list of each fruit. However, a slight modification will give you the same output:

■ **Example 9.11** Python Example Code

```
for fruit in myList:
    print(fruit[0])
```
■

But what if you are required to access every alphabet of every fruit? You cannot do so using a single loop. So here is where nested loops are mostly required:

■ **Example 9.12** Python Example Code

```
myList = ["apple", "banana","cherry"]
for fruit in myList:
     for alphabet in fruit:
         print(alphabet)
```
■

9.4.1 Matrices

Matrix is basically a two dimensional data structure which is used to present and store data in a more presentable way. Lists are very useful when dealing with two dimensional data structures such as matrices as they can easily allow a programmer to store and access data. For example, consider the following matix:

$$\begin{bmatrix} 9 & 2 \\ 7 & 6 \end{bmatrix}$$

This can be stores in lists in the form of lists within list as:
`Matrix1 = [[9, 2]], [[7. 6]]` Now lets suppose you are required to find the determinant of this matrix. To be able do to so, you will have to access every element and then cross multiply and find the difference. Hence to be able to access an element inside another list you use one square bracket followed by another. The first square bracket contains the index of the sub-list from which the element is to be accessed, and the second square bracket contains the index of the element to be accessed from that sub-list. Hence, here is how you can access every element from `Matrix1`

```
Matrix1 = [[9, 2]], [[7. 6]]
int index_1 = Matrix1[0][0]     # Stores 9
int index_2 = Matrix1[0][1]     # Stores 2
int index_3 = Matrix1[1][0]     # Stores 7
int index_4 = Matrix1[1][1]     # Stores 6
print((index_1 * index_4) - (index_2 * index_3))      #evaluates the determinant
```

This code uses data from a two dimensional list to create a pattern:

■ **Example 9.13** Python Example Code

```
data = [[50, 0], [61, 204], [83, 51], [69, 102], [71, 0], [50, 153], [29, 0],
[31, 51], [17, 102], [39, 204] ]
def setup():
     size(100,100)

def draw():
     for i in range(0, len(data)):
        fill(data[i][1])
        rect(0, i*10, data[i][0], 8)
```
■

Exercise 9.1 Write a function to multiply the values from two lists together and return the result to a new array. Print the result to the console ■

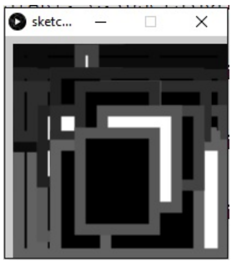


Figure 9.1: Example Sketch

Exercise 9.2

- Create a txt file containing the following data on separate lines: data = 12, 67, 45, 67, 98, 33 Load the data in your program and store it in a list.
- Write a code that chooses any two random numbers from the list `data` and stores them into two different variables. *Use the random function*
- Use the variables as parameters to your rect() function, e.g., rect(var1, var2, var1, var2). Also use the stroke() and stroke Weight() functions.
- Modify your program such that the rectangle fill is black when the count of your draw window is even and white if the count is odd. Your draw() should keep drawing rectangles until you close your program. Sketch 9.4.1 is what your program should look like.

■

Exercise 9.3 Generate a list containing prime numbers in the range 0 - 100. In the draw funtion generate a random number in the given range. If the number exists in your list, draw an ellipse with the parameters of your choice. If the number does not exist in your list, draw an ellipse.

■