

PROGRAMMING FUNDAMENTALS

Python Manual

Copyright © 2018 Habib university

PUBLISHED BY HABIB UNIVERSITY

BOOK-WEBSITE.COM

Licensed under the Creative Commons Attribution-NonCommercial 3.0 Unported License (the “License”). You may not use this file except in compliance with the License. You may obtain a copy of the License at <http://creativecommons.org/licenses/by-nc/3.0>. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

First printing, July 2018




Contents

1	Getting Started	7
1.1	What is Processing?	7
1.2	Installing Processing	7
1.3	Let's Begin	8
1.4	Syntax	9
1.4.1	setup()	9
1.4.2	draw()	10
1.4.3	draw()	10
1.4.4	Basic Shapes	11
1.5	Exercises	11
2	Variables, Conditionals and Print statements	13
2.1	Variables	13
2.1.1	Global and Local Variables	14
2.1.2	Data types	15
2.1.3	Arithmetic Operations	15
2.1.4	Arithmetic operations on different data types	16
2.1.5	Common Operators	18
2.2	Print Statements	18
2.2.1	Console	18
2.2.2	Syntax	19
2.3	Conditionals	19
2.4	Exercises	21

3	Loops	23
3.0.1	Introduction	23
3.0.2	For loop	23
3.0.3	While loop	25
3.0.4	Using for and while loops interchangeably	26
3.0.5	Nested Loops	28
3.0.6	The break statement and continue statements	29
3.0.7	Java break statement	29
3.0.8	The continue statement	29
4	Functions	31
4.1	Introduction	31
4.2	Built in functions	31
4.3	Parameters	32
4.4	Syntax	32
4.5	Transformations	34
4.5.1	Rotations	35
4.5.2	Scale	36
5	Functions(2)	37
5.1	Return values and function statements	37
5.2	Default Arguments	37
5.3	Composition	38
5.4	Recursion	39
6	Strings	43
6.0.1	Creating Strings	43
6.0.2	String Methods	44
6.1	String Indexes	46
6.2	String substring()	46
6.3	Updating Strings	46
7	Data : Files	49
7.1	Introduction	49
7.2	Export files	49
7.3	File Reading	50
8	Arrays	53
8.0.1	Declaring Arrays	53
8.0.2	Instantiating Arrays	54
8.0.3	Accessing Array Elements	54
8.0.4	Array Operations	55
8.0.5	Iterating Arrays	55

8.1	Array Functions	56
8.1.1	append()	56
8.1.2	shorten()	56
8.1.3	expand	56
8.1.4	arrayCopy()	57
8.2	Two-Dimensional Arrays	57



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.

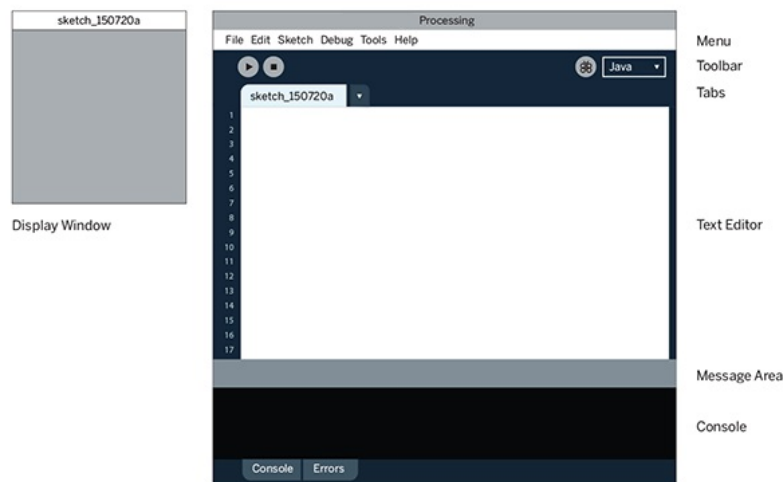


Figure 1.1: The Processing Window

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.

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

```
void setup()
{
    size(480, 120);
}
void 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. The following code for example, will only setup the processing window for you which will be 500

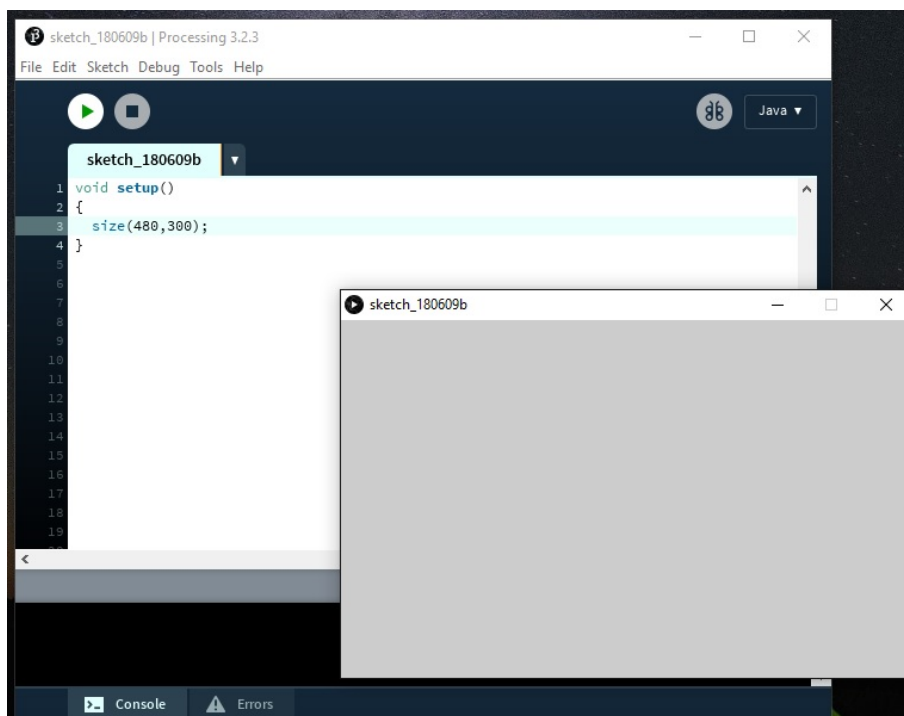


Figure 1.2: The Processing Window

pixels wide and high.

■ Example 1.1 Java Example Code

```
void setup()
{
    size(500,500);
}
```

`void` is the keyword for initiating those functions in Java that do not return any value. It will always be followed by the name of your function and parenthesis 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, return values and their syntax will be covered in detail in chapter 4.

1.4.2 draw()

1.4.3 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.

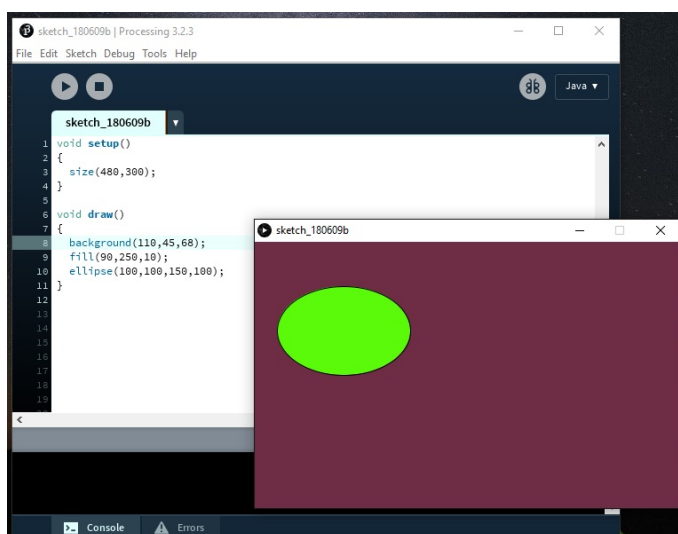


Figure 1.3: The Processing Window

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.

1.4.4 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 `int x=23`, then x is the variable and 23 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.

In every computer program, it is important to declare the variable before it is used. It is also important to specify the data type of the variable, for example, if you're storing an integer, it is important to tell your program that it is an integer data type. Data types are further explained in the coming sections.

■ Example 2.1 Java Example Code

```
String 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 Java Example Code

```
void sum()
{
    int x = 3;
    int y = 4;
    int 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
```

■

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 Java Example Code

```
void setup()
{
    size(500,500);
}
void draw():
    int 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

```
void setup()
{
    size(500,500);
}
int x = 0;
void draw():
{
    background(x);
    x = x + 1;
    println(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 bits(1s or 0s) making it necessary for your system to differentiate between the two. Different values are stored differently, for example, 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 example, 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 example, `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.
- **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. char literals are surrounded by single quotes e.g. `char x = '!';`
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.
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. 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.

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 Code

```

void setup()
{
    size(500,500);
}
void 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 are 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.

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

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. If you add two float values, your program will return a float value. Similarly, operation on an integer and a float will result in a float value.

```

println(4/3); // Prints 1
println(4.0/3); // Prints 1.333334
println(4/3.0); // Prints 1.333334
println(4.0/3.0); // Prints 1.333334

```

However, if you assign two different data types to two different variables then you cannot perform arithmetic operations on them, for example, you cannot add a float variable to an int variable.

The following examples will help you understand how different data types work.

```

int a = (4/3); // Assign 1 to a
int a = (3/4); // Assign 0 to b
int a = (4.0/3); // Error!
int a = (4.0/3.0); // Error!

```

Now look at the following

```

float a = (4.0/3); // Assign 1.333334 to a
float a = (4.0/3.0); // Assign 1.333334 to a

```

The reason why the last two examples do not return an error is because it is possible to convert an

integer value to a float value but not vice versa. Hence, the final output is converted to float after the calculation has been performed.

```
float a = (4/3); // Assign 1 to a
float a = (3/4); // Assign 0.0 to a
```

The following code multiplies 2 float values and returns a float value:

■ Example 2.5 Java Example Code

```
float x = 3.5;
float y = 9.0;
print(x * y);
```

Output:

```
» 31.5
```

■

Although you cannot perform mathematical operation on two different types but try this!

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

Converting Data types

In the previous section we saw how your program can directly convert an int data type to a float, but a float can't be converted to an int data type directly. Hence, such a case requires explicit conversion.

```
float f = 12.6;
int i = 127;
f = i; // converts 127 to 127.0
i = f; // Can't automatically convert a float to an int
```

Here is how data conversions work:

- *boolean()*: The *boolean()* function converts 0 to False and all other numbers to 1.

```
int i = 0;
boolean b = boolean(i); // Assigns False to b
```
- *char()*: The *char()* function converts other types of data to character representation. To understand char read more about ASCII codes at <http://www.asciitable.com/>

```
int i = 65;
char c = char(i) // Assigns 'A' to c
```
- *float()*: Converts numbers to floating-point representation.

```
int i = 2;
int j = 3;
float f1 = i/j; // Assign 0.0 to f1
float f2 = i/float(j); // Assign 0.6666667 to f2
```
- *int()*: Converts other types of data to integer data type.

```
float f = 65.9;
int i = int(f); // Assign 65 to i
char c = 'E';
i = int(c); // Assign 69 to i
```

2.1.5 Common Operators

Logical operators applicable in java are `&&` , `||` and `!` which help you to construct compound conditions. `&&` is used for **and** operation, `||` is used for **or** operation while `!` is used for not operation. 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
<code>==</code>	Indicates equality. If two values are equal then the condition is True	<code>"a" == "a" » True</code>
<code>!=</code>	If two values are not equal, then the condition is True	<code>3 != 4 » True</code>
<code><</code>	Left operand being less than the right operand, returns True	<code>3 < 4 » True</code>
<code>></code>	Left operand being greater than the right operand, returns True	<code>4 > 3 » True</code>
<code><=</code>	Left operand being less than or equal to the right operand, returns True	<code>3 <= 4 » True</code>
<code>>=</code>	Left operand being greater than or equal to the right operand, returns True	<code>4 >= 3 » True</code>

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.

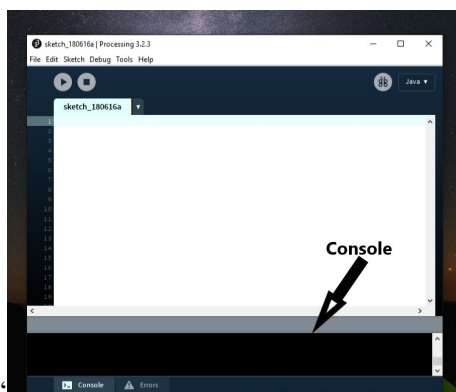
Exercise 2.3 Now initiate 2 more variables and assign them a value of 1. Give a color to your rectangle using fill(). Use your variables as parameters for fill() and keep incrementing your variables.

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.

The following 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.

```
void setup()
{
    size(500,500);
}
int x = 0;
void draw()
{
    background(x);
    x = x + 1;
    println(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.

The following code prints PROCESSING in the console area.

■ Example 2.6 Java Example Code

```
void setup()
{
    size(500,500);
}
void draw()
{
    println("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.

Syntax

```
if (condition)
{
    statements
}
```

Let's suppose that want your program to print the value of a variable 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 2.7** Java Example Code

```
void setup()
{
    size(500,500);
}
void draw()
{
    number = int(random(90));
    if (number%2==0);
    {
        println("EVEN");
    }
    else
    {
        println("ODD");
    }
}
```

■

Take another example, Suppose you want you program to print the value of a variable if it's even and greater than 100. In this case you need compound conditions. which require logical operators. Look at the following example code:

■ **Example 2.8** Java Example Code

```
void setup()
{
    size(500,500);
}
void draw()
{
    number = int(random(90));
    if ((number%2==0) && (number>100));
    {
        println("EVEN");
    }
    else
    {
        println("ODD");
    }
}
```

■

Here, the ampersand in the if condition is used to set up a compound condition where your program is required to fulfill both of the conditions to print "EVEN" on your console.

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

3.0.1 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 your 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.

3.0.2 For loop

The for loop is used for lines of code that are to be implemented in a sequence. The for loop takes range as parameter and iterates over your input only in the specified range. A for loop is more precise as compared to a while loop as it groups all the individual statements of a while loop inside parenthesis.

Syntax

```
for (start; end; step)
{
    statements
}
```

Now let's suppose I want to print my name 50 times on the console. It is possible to do so using **println("Kainat")** written 50 times on my editor, but writing the same statement 50 times would be a chaos and is an inefficient way of programming. So, I can simply use a loop that will do the same job in a very few lines.

Here is a simple for loop that prints my name 50 times on the console:

■ Example 3.1 Java Example Code

```
for (inti = 0; i > 50; i++)
{
```

```
println("Kainat")
}
```

The code in the next example prints numbers from 0 to 9 on your console.

■ Example 3.2 Java Example Code

```
for (int i=0; i<10; i=i+1)
{
    print(i);
}
```

■

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 instead *i* and you would still have the same output.

Code that has to be repeated is placed inside the block of the for loop. *The lines of code grouped together by enclosing them between curly braces form a **block**.*

`print(i)` is placed inside the block to print the value of *i* at every iteration.

This code would print the entire output on the same line. To avoid printing on the same line you could use `println(i)`.

*If you want a comma after every number you can use `print(i, ",")` to put a comma every time your code prints *i*.*

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. The for loop requires a lower limit, a higher limit and a 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 (int i=0; i <= 100; i = i + 10)
```

then, the value of *i* begins at 0, steps 10 numbers and assigns *i* a value of 10. In this case your output will be *10,20,30,40* and the program will terminate as *i* reaches 100.

You can make your for loop run in the reverse order too. For example,

```
for (int i=100; i>=0; i=i-10)
```

will give *i* an initial value of 100 and terminate the program in 10 iterations as your *i* reaches 0.

Now let's suppose you have a string that you wish to iterate and print its alphabets one by one. Java does not allow you to access the elements of a string directly. To access the elements of a string in java, you may use **`charAt(int index)`** method. This function returns the character at the specified index. The following example code accesses the elements of a string in Java.

■ Example 3.3 Java Example Code

```
String s = "Processing";
for (int i=0; i<s.length(); i++)
{
    char c = s.charAt(i);
    print(c);
}
```

■

`length()` is a built-in function which stores the length of your string, list or any other type of data. If you have a variable `a='Processing'`, then `a.length()` will store 10 as there are 10 characters in the word.

In this example, the loop iterates over the length of your variable, and `string.charAt(i)` takes the index `i`, accesses the character on that index and prints it to the console. At the first iteration the loop accesses the letter "P", on the second iteration it accesses the letter "r" and so on. Hence, this method is easier to directly access elements in a string.

The output of this code would be the letters of "processing" printed on the same line.

■ **Example 3.4** 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.

```
void setup()
{
    size(500,500);
}
void draw()
{
    for (int i=79; i<=0; i=i-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 rewrite `ellipse()` with different parameters over and over again to get the same output.

Your initial ellipse would 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. In the code above, your program initiates the value of `i` from 79 which runs back to 0, skipping 3 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?

Try running the loop oppositely and see if you're able to get the same pattern. ■

Exercise 3.1 Use the code in example 3.4 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. ■

3.0.3 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 also has three parts. In a for loop you give `i` (any randomly chosen alphabet) an initial value from which it 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. All of these tasks are accomplished in one line in a for loop while in a while loop you will have to specify all three in different steps. Our example codes from section 3.2 can easily be implemented using while loop as follows:

■ Example 3.5 Java Example Code

```
int 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 3.2.

Line # 1 initiates and assigns an initial value to 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.

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.

■ Example 3.6 Java Example Code

```
int number=1
while (number<100)
{
    number=number*2
    println(number)
}
```

■

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

3.0.4 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 be 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 3.7 Java Example Code

- **for loop**

```
for (int i=100; i<=10; i=i-1)
{
    print(i);
}
```
- **while loop**

```
int 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, for loop is more appropriate to use in this case. ■

■ Example 3.8 Java example code

- **while loop**

```
int i=1;
int factorial=1;
while (i<=number)
{
    factorial=factorial*i;
    print(factorial);
}
```
- **for loop**

```
factorial=1;
for (int i=0; i<=number; i=i+1)
{
    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/>

3.0.5 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 3.9 Java Example Code

```
for (int i=0; i<=5; i=i+1)
{
    for (int j; j<=10; j=j+1)
    {
        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 3.10 Java Example Code

```
for (int i=10; i<=100; i=i+1)
{
    for (int j=10; j<=100; j=j+1)
    {
        point(x,y);
    }
}
```

Here is another example followed by the output:

■ Example 3.11 Python Example Code

```
noStroke();
for (int y = 0; y < 100; y+=100)
{
    for (int x = 0; x <100; x +=10)
    {
        fill(x + y) * 1.4);
        rect(x, y, 10, 10);
    }
}
```

Exercise 3.4 Using textbfnested 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.* ■

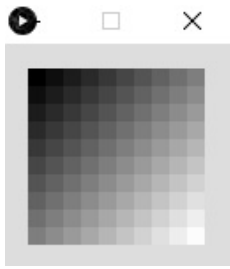


Figure 3.1: Example Sketch

3.0.6 The break statement 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. Thus, the break and continue statements are used to alter the normal flow of a loop.

3.0.7 Java break statement

The break function is used to immediately let a programmer break out of a loop and enter the next condition(if any). 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. Here is how it works:

■ Example 3.12 Java Example Code

```
String s = "Processing";
for (int i=0; i<s.length(); i++)
{
    char c = s.charAt(i);
    if c == 's'
    {
        break;
    }
    else
    {
        print(c);
    }
}
```

In the example above the loop is initially set to run until it completely iterates the string, but since the condition breaks the loop on the character 's', therefore it only runs as long as it does not reach 's'.

3.0.8 The continue statement


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

■ Example 3.13 Java Example Code

```
String s = "Processing";
```

```
for (int i=0; i<s.length(); i++)  
{  
    char c = s.charAt(i);  
    if c == 's'  
    {  
        continue;  
    }  
    else  
    {  
        print(c);  
    }  
}
```

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 its 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, you have seen that 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()` function are functions that you initiate on your window and they execute any lines of code that you write. The `line()` and the `ellipse()` function 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

```
void setup()
{
    size(500, 500);
}
void 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 its 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 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 Java, you initiate a function using the data type it is expected to return. 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 data type followed by the name of your function, further followed by parenthesis as shown below.

```
void functionName()
```

The function named 'functionName' here takes no parameters and returns nothing.

```
void newFunction(x1,x2)
```

`newFunction()` here takes 2 parameters `x1` and `x2`.

Consider the following example:

■ Example 4.2 Java Example Code

```
void setup()
{
    size(100, 100);
    noStroke();
}
void 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 since we know that `draw()` is usually used for lines of code that need to repeat themselves to create animation. Hence, we can have another function for drawing an eye:

■ Example 4.3 Java Example Code

```
void setup()
{
    size(100, 100);
    noStroke();
}
void draw()
{
    background(204);
    eye(65, 44);
}
void 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:

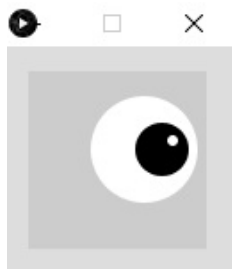


Figure 4.2: Example Sketch

Exercise 4.1 Modify example 4.3 such that it draws a pair of eyes instead of a single eye.
hint: You just need to call the eye function twice using different parameters ■

4.5 Transformations

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 following example is how translation works.

■ Example 4.4 Java Example Code

```
void setup()
{
    size(500,500);
}
void 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 Java Example Code

```
void setup()
{
    size(500,500)
}
void draw()
{
    for (int i=0; i<=20; i++)
    {
        translate(20,10);
        rect(0,0,100,100);
    }
}
```

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.

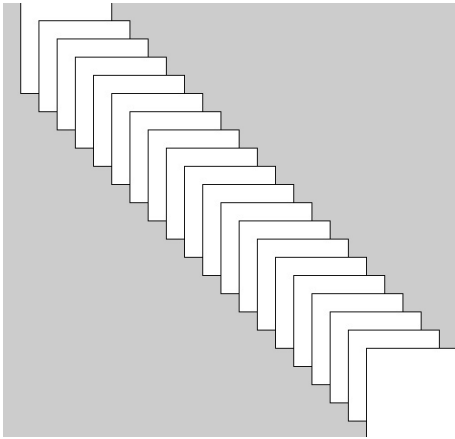


Figure 4.3: Example Sketch

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 its 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.* ■

4.5.1 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 Java Example Code

```
rect(55,0,30,45);
rotate( $\frac{\pi}{8}$ ); #clockwise rotation
rect(55,0,30,45);
```

■

■ Example 4.7 Java Example Code

```
translate(45,60);
rect(-35, -5, 70,10);
rotate(- $\frac{\pi}{8}$ );
rect(-35, -5, 70, 10);
rotate(- $\frac{\pi}{8}$ );
```

```
rect(-35, -5, 70, 10);
```

■

4.5.2 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 Java Example Code

```
ellipse(32, 32, 30, 30);  
scale(1.8); # scales the shape to 180%  
ellipse(32, 32, 30, 30);
```

■

■ Example 4.9 Java 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)

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 that calculates the product of two numbers and returns the output. The output may be float or integer. Your return statement will immediately exit the function without executing any lines of code further. Functions that do not return any value, for example, functions that are only required to print on the console or draw some shapes do not return any value and are considered as the special type `void()`. `void()` is specially used to declare such 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 following example makes calculations and returns 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 Java Example Code

```
void setup():  
    float c = fahrenheitToCelcius(451.0)  
    println(c)  
float fahrenheitToCelcius(float t):  
    float f = (t-32.0) * ( $\frac{5.0}{9.0}$ )  
    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

```
void 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** Java Example Code

```
int val
int Calculate(int a, int 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 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 enable 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 to call 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. If you do not specify the stopping condition, the function will keep calling itself infinitely and your program will return you an error. 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 Java Example Code

```
void setup()
{
    rec_exp();
}
void rec_exp()
{
    print("Processing");
}
```

Output:

»Processing

■

■ Example 5.5 Java Example Code

```
void setup()
{
    rec_exp();
}
void rec_exp()
{
    print("Processing");
    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** Java Example Code

```
void setup()
{
    rec_exp(10);
}
int count = 0;
void rec_exp(int 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** Java Example Code

```
void setup()
{
    print(fact(10));
}
int fact(int n)
{
    if (n <= 1) base case
    {
        return 1;
    }
    else
    {
        return n*fact(n-1);
    }
}
```

■

The else statement in this function calls the function again and again until the base case is

reached.

Exercise 5.2 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** Java Example Code

```
void setup()
{
    size(100, 100);
    drawLines(5, 15);
}
void drawLines(int x, int num)
{
    line(x, 20, x, 80);
    if (num > 0)
    {
        drawLines(x + 5, num - 1);
    }
}
```

Output:

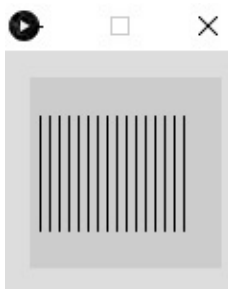


Figure 5.1: Example Sketch

Exercise 5.3 Write a function to draw a shape of your choice. Use it to draw the same shape to the screen multiple times, each at a different position using Recursion. ■

Exercise 5.4 Rewrite example 5.8 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. For example, 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 within double quotes in java creates a `String`. 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, however, are declared in a similar way as in other data types:

```
String 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 its 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.*

Syntax

The syntax for assigning a string to a variable is as follows:

```
String string_name = "String";
```

6.0.1 Creating Strings

There are several ways to create string in Java. The simplest method is by using the string literal as:

```
String string_1;
```

Other methods are as follows:

- Using the new keyword:

```
String string_1= new String ("Processing in Java");
```

- Using another string:

```
String string_1 = new String(string_2);
```

- Using Concatenation:

```
String string_1 = "Welcome" + "to" + "Strings";
```

6.0.2 String Methods

length

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

■ Example 6.1 Java Example Code

```
String string1 = "Processing";
```

```
String string2 = "hi";
```

```
print(string1.length())
```

Output:

```
»10
```

```
print(string2.length())
```

Output:

```
»2
```

■

toLowerCase()

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

■ Example 6.2 Java Example Code

```
String string1="PROCESSING";
```

```
print(string1.toLowerCase())
```

Output:

```
» processing
```

■

toUpperCase()

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

■ Example 6.3 Java Example Code

```
String string1="processing";
```

```
print(string1.toUpperCase())
```

Output:

```
»PROCESSING
```

■

trim()

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

■ Example 6.4 Java Example Code

```
var = "      Processing      "
```

```
print(var.trim())
```

Output:

» Processing ■

split()

The **split()** method divides your string into its separate elements by forming a list of those elements.

■ Example 6.5 Python Example Code

```
String str = "P-r-o-c-e-s-s-i-n-g";
String[] array_string = str.split("-", 10);
print(array_string);
```

Output: "P r o c e s s i n g" ■

■ Example 6.6 Python Example Code

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

Output:

» "hello world" ■

compareTo()

This method compares two strings directly and lets you know if they are equal. The method returns 0 if two strings are equal and a negative or a positive value otherwise. The positive value indicates that the first string is lexicographically greater than the second other and vice versa in case of a negative value.

This method is case sensitive, you can use **compareToIgnoreCase()** instead of **compareTo()**.

■ Example 6.7 Java Example Code

```
String string_1 = "Programming";
print(string_1.compareTo("is easy"));
```

Output:

» 7 ■

■ Example 6.8 Java Example Code

```
String string_1 = "Processing";
print(string_1.compareTo("Processing"));
```

Output:

» 7 ■

Read more about String methods at https://www.tutorialspoint.com/java/java_strings.htm

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 Indexes

The `charAt` method allows you to access string elements, i.e, individual characters.

```
string.charAt(int index)
```

■ **Example 6.9** Java Example Code

```
String string1 = "Hello World";
print(string1.charAt[4]); //prints the letter at position 4
Output :
» 0
```

■

It is also possible to access more than 1 character using the string substring method discussed in the next section.

6.2 String substring()

This method allows you to obtain a substring from a given string. `substring` takes 2 arguments which are the starting and the ending index of the substring you want. If in any case you pass only one argument, this function would give you a substring from the specified index till the end of your string.

```
String substring(int beginIndex, int endIndex);
```

■ **Example 6.10** Java Example Code

```
String string_1 = "Learning to code in Java";
print(string_1.substring(0, 16));

Output:
»Learning to code
```

■

6.3 Updating Strings

Since strings are immutable, i.e, you can't make changes to your string once it is created. You can, however, make changes to your original string by updating your strings. One way to do so is Concatenation. String class provides you a method for concatenating two strings.

```
string1.concat(string2)
```

```
// Concatenates string2 to string1
```

■ **Example 6.11** Java Example Code

```
String var = "Hello";
print(var.concat("World"));
Output:
»HelloWorld
```

■

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

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. Data : 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 its 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 this example code:

■ Example 7.1 Java Example Code

```
PrintWriter output;
void setup()
{
    size(500,500);
    output = createWriter("newfile.txt");
}
int count = 0;
void draw()
{
```

```

    count = count + 1;
    output.println(count);
}

void keyPressed()
{
    output.flush();
    output.close();
    exit();
}

```

Here is how it works:

We initiate a file named "newfile" and assign it to a variable output.

The print function 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 program 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:

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 (int i ==0; i<=lines.length(); i++)  
{  
    println(lines[i]); // where your program prints the element at the index as i increases.  
}
```

 ■



8. Arrays

Arrays are objects that allow for structured grouping or sequential collection of data of the same type. The data type maybe any data type, such as int, float, byte, double, char, boolean or even a **class**. Data that an array stores is basically stored as a variable in the array but you do not need to declare every variable separately. Every variable within an array has the same name grouped with it's index. For e.g, if you name your array as myArray, then to extract the element at the first position you'd use myArray[0] where 0 is the index of the first first variable in myArray. Arrays are useful for operations on grouped data, such as data stored in tables or working with matrices can be very convenient using arrays.

8.0.1 Declaring Arrays

There are more than one way to declare an array in java, but we will focus on one of the most commonly used and preferred method. To declare an array you first declare the data type of the elements that you wish to collect, followed by square brackets and the array name as:

```
int[] myArray;
```

Another way to declare an array is

```
int myArray [];
```

If you wish to store your favorite songs in an array, you may declare your array using the data type String as:

```
String [] favSongs;
```

Boolean Arrays can be declared as:

```
boolean [] arrayName;
```

There are a few other methods too which are preferred while working with classes and objects. We will be considering only the first method while understanding arrays.

In Processing you declare arrays at the very start of your program i.e before the setup() function.

8.0.2 Instantiating Arrays

Java Arrays are objects that do not allow you to specify the **size** of an array. Declaring an array only allows you to specify the data type and name of your array. So, you do not allocate memory for an array while declaring it. To do so, you will have to instantiate an array using the keyword **new**. This is how instantiating an array works:

```
arrayName = new datatype [size of your array];
```

For an array of your favorite songs containing 5 elements:

```
favSongs = new String [5]
```

When you instantiate an array all elements of your array are given an initial value for e.g. for a boolean Array all elements of your array are initially set to the default value *False* and for an integer/long/short/byte all elements are given an initial value of 0. Similarly char elements are set to Unicode null character and object references are set to null.

Arrays can be instantiated while declaration. The syntax for this is as follows:

```
datatype [] arrayName = new datatype [size];
```

Or you may separately declare and instantiate as :

```
datatype [] arrayName
arrayName = new dataType [size of the array]
```

You may instantiate an array by assigning initial values to while declaring an array. For this you do not specify that **size** of the list neither use new because the **size** of your array is the number of the elements in your list. This is done by using a comma separated list of your initial values enclosed in curly brackets:

```
datatype [] arrayName = {comma separated list of your initial values}
String [] favMovies = { "Despicable Me" , "Minions", "Moana", "Coco" }
```

8.0.3 Accessing Array Elements

To be able to perform multiple operations on an array you need to access its elements. For e.g. to print the elements of an array you need to access the elements one by one and then print them since java does not support aggregate operations such as printing or copying arrays . You may access an element in your array by specifying the name of your Array followed by square brackets enclosing the index or position number of the element you want to access.

The syntax to access an element of an array is:

```
arrayName [index]
```

Consider the array *myArray* that contains a list of your favorite movies:

```
String [] myArray = ["Despicable me", "Minions", "Moana", "Coco"];
```

If you're required to get a movie at the 3rd position then the syntax would be:

```
print(myArray[index - 1];)
```

i.e

```
print(myArray[2];)
```

To get an element at the last position:

```
myArray[ myArray.length - 1]
```

where array.length() is the method to get the length of your array.

8.0.4 Array Operations

Java does not support aggregate operations on Arrays. For e.g, you cannot use `print(Array)` to print your array on the console, or copy an array to another array or adding elements to your array. Processing however allows you some of these operations. For e.g, you can print an array using `printArray(ArrayName)`; to display the contents of your array on the console. But it does not support a lot of other operations which Java does not support. To be able to perform those operations you'll have to take a different approach for e.g, if you do not use `printArray()`; to print your array you may write your own function that iterates over your array and prints every element individually. In the coming sections we will cover iterating arrays to access array elements and use them for your program.

8.0.5 Iterating Arrays

After you declare an array, you may access individual elements of the array using the syntax that we discussed in the previous sections. The following code uses the elements of an array to draw a pattern using the `line` function:

■ Example 8.1 Java Example Code

```
int[] data = {19, 40, 75, 76, 90}
line(data[0], 0, data[0], 100);
line(data[1], 0, data[1], 100);
line(data[2], 0, data[2], 100);
line(data[3], 0, data[3], 100);
line(data[4], 0, data[4], 100);
}
```

Your code will return an `ArrayIndexOutOfBoundsException` if you try to access the 5th element because although the total number of elements in your array is 5, index of the last element is 4 as the first index is at position 0.

The code above can be made efficient by using loops. Using loops, there will no longer be a need to access every element individually as the loop will do that for you:

■ Example 8.2 Java Example Code

```
int [] data = {19, 40, 75, 76, 90}
for (int i = 0; i<data.length; i++)
{
    line(data[i], 0 ,data[i], 100)
}
```

You may also use a for loop to assign values to your array as in the given example:

■ Example 8.3 Java Example Code

```
int[] data;
data = new int[10]
for (int i=0; i<10; i++)
{
    data[i] = i+2
}
printArray(data);
```


The next example draws a pattern of rectangles with a collection of data of it's parameters using loops to iterate over your array :

■ **Example 8.4** Java Example Code

```
int[] x = {50, 61, 83, 69, 71, 50, 29, 31, 17, 39};
fill(0);
for (int i = 0; i < x.length ; i++)
{
    rect(0, i*10, x[i], 8)
}
```

■

8.1 Array Functions

There are a number of array functions that processing provides. A few of them are listed below.

8.1.1 append()

This function allows you to add one element at the end of your original array.

■ **Example 8.5** Java Example Code

```
String trees = {"ash","oak"};
trees = append(trees, "maple");
printArray(trees);
Output: ["ash", "oak", "maple"]
```

■

In the example above line #2 adds "maple" at the end of your original array and assigns trees to that variable.

8.1.2 shorten()

This function works exactly opposite to the append() function. It shortens your array by one element by removing the last element of your array.

■ **Example 8.6** Java Example Code

```
String[] trees = {"ash", "oak", "maple"};
trees = shorten(trees); printArray(trees);
Output: ["ash", "oak"]
```

■

8.1.3 expand

Although the append() function allows you to add elements to your array, the expand function also allows you to increase the [size](#) of your array. In fact, for adding more than one element to your array, the expand() function is more efficient than the append() function. expand() can increase your array to a specified [size](#) or double the [size](#) if you do not specify the [size](#).

■ **Example 8.7** Java Example Code

```
int [] x = new int[100];
int count = 0;
void setup()
{
    size(100, 100);
}
void draw()
{
    x[count] = mouseX;
    count++;
    if (count == x.length)
    {
        x = expand(x);
        println(x.length)
    }
}
```

This code expands your array by doubling its **size** as soon as the value of count becomes equal to the **size** of your original array.

8.1.4 arrayCopy()

Sometimes a programmer may want to copy elements of one array to another. To do so, one way could be to copy every element individually by accessing each element using a loop. The other one is to use the built-in `arrayCopy()`. This function will copy the entire contents of your array to another array.

This function will work only if both of your arrays are of the same **size**. It takes 2 arrays as parameters and copies the contents of the first array to the second array.

■ **Example 8.8** Java Example Code

```
String[] north = {"OH", "IN", "MI"};
String[] south = {"GA", "FL", "NC"};
arrayCopy(north, south);
printArray(south)
Output : ["OH", "IN", "ME"]
```

8.2 Two-Dimensional Arrays

The concept of arrays is very important when dealing with tables, charts or matrices. To be able to perform mathematical operations on data present in these forms, one may need to store data in such a way that it is easier to access and understand. 1 dimensional arrays work mostly like strings. With little efficiency, every program using 1d arrays can be re-written using strings. However, data in the form of tables or matrices can only be stored and accessed using multidimensional arrays.

Consider the following data:

50	61	83	69	71	50	29
0	204	51	102	0	153	0

The syntax for 2D arrays is the same as for 1D array with a few minor changes. For e.g the

following code stores the data in table 8.2 and accesses it:

■ **Example 8.9** Java Example Code

```
int[] [] x = {{50,0} , {61,204}, {83, 51}, {69, 102}, {71, 0}, {50,153}, {29,0}
} Accessing elements of an array works exactly as accessing elements in strings. Square brackets
containing the index of the element to be accessed followed by another square bracket containing
the index of the sub-element inside.
println(x[0][0]) prints "50"
println(x[0][1]) prints "0"
println(x[3][0]) prints "69"
println(x[5][1]) prints 153 ■
```

Matrices

Multidimensional (2D) arrays play an important role while dealing with matrices. It is very convenient to display matrices in the form of arrays to be able to perform mathematical operations on them. For e.g, the following matrix can be displayed in the array as:

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

Matrix1 = {{9, 2}, {7, 6}} Here is how you can find the determinant of the matrix displayed above:

■ **Example 8.10** Java Example Code

```
int [] 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:

Exercise 8.1 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 ■

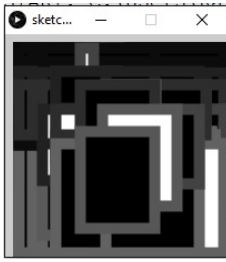


Figure 8.1: Example Sketch

- Exercise 8.2**
- 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 8.2 is what your program should look like.

- Exercise 8.3** Generate a list containing prime numbers in the range 0 - 100. In the draw function 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, append it to your list.