Arrays

So far, we have seen how to create variables in our processing programs that can be used to store simple values (or primitive types):

```java
int x = 0;

float delta = 0.483;
```

In the above definitions, the variables are each associated with a single value (defined by the type `int` or `float`). We have also seen instances where multiple values were associated with a single variable name:

```java
color darkOliveGreen = (85, 107, 47);
String colorName = "Dark Olive Green";
PImage castle = loadImag("myCastle.jpg");
```

The types `color`, `String`, and `PImage` each associate a number of values with the variable being defined. The meaning of the values is determined by the type definition itself. Thus, any variable of type `color` will have three RGB values associated with it, a variable of type `String` can be a string of characters, and a variable of type `PImage` associates with itself all the pixel values of an image. In programming language terminology, we distinguish between `simple types` (like `int`, `float`, etc.) that associate a single value with its variables and `compound or complex types` (like `color`, `String`, `PImage`, etc.) that may have several values aggregated together in a single variable. Most modern programming languages provide facilities for creating new aggregate types which we will see later.

Data or values can also be aggregated into `arrays` which are container structures that enable storage of many values of the same type together in a single variable. For example, consider the set of numbers in the table below:

<table>
<thead>
<tr>
<th>Petroleum</th>
<th>Coal</th>
<th>Natural Gas</th>
<th>Nuclear</th>
<th>Renewable</th>
<th>Hydropower</th>
</tr>
</thead>
<tbody>
<tr>
<td>40.0</td>
<td>23.0</td>
<td>22.0</td>
<td>8.0</td>
<td>4.0</td>
<td>3.0</td>
</tr>
</tbody>
</table>

These numbers represent the percentage of United States national energy consumption listed by energy source in 2005. For example, 8% of the overall energy consumed came from nuclear energy sources. It would be easy to create six variables to store each of the values:

```java
float petroleum = 40.0;
float coal = 23.0;
float naturalGas = 22.0;
float nuclear = 8.0;
float renewable = 4.0;
float hydropower = 3.0;
```

However, for related values like these that form a dataset it is more convenient to aggregate or group them into a single container, called an `array`. Arrays aggregate as well as structure datasets like these and make them available for use through a single named variable (as opposed to six in the case above). Each individual value is accessed by indexing it in the array. A better way to visualize this representation is as follows:
In Processing, the above array can be defined as:

```java
float[] consumption;
```

It defines a variable called `consumption` to be an array whose values will be of type `float`. Notice that it does not define the size or the number of elements it holds. To specify the size of the array you have to further issue the following command:

```java
consumption = new float[6];
```

This command is a request to create a new array that can store 6 `float` values. It is possible to combine the above two commands:

```java
float[] consumption = new float[6];
```

Once you have defined, and created the array, you can store values in it:

```java
consumption[0] = 40.0;
consumption[1] = 23.0;
consumption[2] = 22.0;
consumption[3] = 8.0;
consumption[4] = 4.0;
consumption[5] = 3.0;
```

There is a better way to do this where you can define, create, and initialize the data in an array in a single command:

```java
float[] consumption = {40.0, 23.0, 22.0, 8.0, 4.0, 3.0};
```

That is, you are defining an array of type `float`, it is called `consumption`, and it has the six specified `float` values placed in it. This is how you define, create, and initialize an array in a single command. If you look carefully, the format for array initialization is not that different from that of initializing simple variables:

```java
float salary = 1000000.00;
```

The only difference is that you have to specify all the aggregate values of the array by placing them inside curly braces. Processing will count the number of values you specify and then implicitly create an array of exactly that size (6 in the case above). The following summarizes all the ways arrays can be defined, created, and/or initialized:

```java
TypeName[] arrayName;
// defining an array variable to hold values of TypeName[] arrayName = new TypeName[N];
// creating an array of size N

TypeName[] arrayName = new TypeName[N];
// defining and creating an array of size N
```
TypeName[] arrayName = {v1, v2, …, vN};
// defining, creating, and initializing an array of size N+1

Here are some more examples of array definitions:

// An array to hold the names of all the days in a week
String[] weekDays = {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"};

float[] highTemps, lowTemps;
// two arrays, each containing high and low temperature values

int[] count;     // an array of integers
PImage[] photos;   // an array of photos

// An array to hold the names of months in a year

// An array of famous mathematical constants.
// How many can you recognize?
float[] mathConstants = {3.14159, 2.71828, 1.61803, 1.41421, 1.732, 0.57721, 1.32471, 0.66061, 0.76422, 0.56714};

// The colors in a rainbow
color[] rainbow = {color(255, 0, 0), color(255, 127, 0),
                    color(255, 255, 0), color (0, 255, 0),
                    color (0, 0, 255), color (111, 0, 255),
                    color (143, 0, 255)};

// Names of various energy sources
String[] energySource = {"Petroleum", "Coal", "Natural Gas", "Nuclear", "Renewable", "Hydropower"};

**Try This:** For each of the array definitions above, clearly identify the parts that make up an array: its variable name, the type of values it holds, the number of values it can hold (its size), if known, and the values it holds, if known.

**Indexing, Size, and Loops**

Individual elements in the array are accessed by specifying their index using the syntax:

arrayName[index]
Thus, to access the string value "Natural Gas" from the `energySource` array shown above we will write:

`energySource[2]`

Remember that individual values or items stored in an array are indexed starting from 0 (for the first element) to N-1 (for the last element). Thus, an array of size 6 has its index in the range 0..5.

Processing also defines a `length` attribute for array variables that can be used to find out the size of an array. Thus, the expression

`energySource.length`

represents the value 6 since, in the definition above, `energySource` was created with six elements. The length attribute is very useful in designing general purpose functions that operate on arrays as we will see below.

Most operations on data stored in an array rely on loops to systematically visit each element of the array. For example, if you wanted to store the number 42 in every location of a 1000 element array:

```java
int[] n = new int[1000];
for (int i=0; i < n.length; i++) {
    n[i] = 42;
}
```

the for-loop enables an easy mechanism to specify a repetitive set of steps succinctly. The loop control variable, `i` also acts as an index into the array. Notice that the initial value assigned to `i` is 0 (the index of the first element of the array) and loop’s terminating condition is `i < n.length`. That is, as soon as the value of `i` becomes equal to `n.length` the loop will terminate. You could alternately specify the for-loop as:

```java
for (int i=0; i <= n.length-1; i++) {
    ...
}
```

In computing, there is a well known error that even seasoned programmers tend to make. It is called **off-by-one error**: whenever the number of iterations specified is off by 1. Here are two examples of off-by-one errors:

```java
for (int i=0; i <= n.length; i++) {
    ...
}
```

```java
for (int i=0; i < n.length-1; i++) {
    ...
}
```

In the first instance, you are specifying the value of `i` to go from 0 to `n.length`, including the value of `n.length`. If, in the loop, `i` is being used to index into the array `n` (as in `n[i]`) the program will generate an "**ArrayOutOfBoundsException**" error. In the second instance, if `i` is being used to access all elements of the array `n`, it will never reach or do anything with the last element in the array (`n[n.length-1]`) since the loop will terminate before getting there. In this case, the program will run without reporting any errors resulting in incomplete computation(s) and incorrect results, aka **bugs**. This is a devious situation that can be hard to debug. Always be sure to review and correct all potential off-by-one errors at the time of writing the program!

As another example, if you wanted to store the outcomes of 10,000 rolls of a six-sided die:
```java
int[] outcome = new int[10000];
for (int i=0; i < outcome.length; i++) {
    // fill it up with random values
    outcome[i] = int(random(1,7));
}
```

When called with two arguments, the function `random(n1, n2)` returns a floating point value in the interval \([n1..n2)\). Thus `random(1, 7)` will generate a value in the interval \([1.0..7.0)\), not including 7.0 itself. In order to generate a number that represents an outcome of throwing a dice, we convert the number into an integer using the `int()` function.

When accessing all elements in an array you typically need an index variable (like `i` above) that starts from the first index (`i = 0`) and goes all the way to the end (`i < outcome.length`), incrementing each time by 1 (i++). If you visualize, in your head, the array laid out with the first item on the left and stretching out across the page, the index travels from left to right as it allows you to visit each element in succession. For our right-leaning readers, the above loop can also be written as:

```java
for (int i=outcome.length-1; i >= 0; i--)  {
    // fill it up with random values
    outcome[i] = int(random(1,7));
}
```

Functionally, the two loops serve exactly the same purpose (i.e. filling up the array with random values in the interval \([1..7))\). In the latter case, you start at the last index (`i=outcome.length-1`) and go all the way to the first element (`i >= 0`) decrementing by 1 each time (i--). Also, in both cases, we make use of the length attribute to identify how far (in the first loop) the index variable should go, or where it should start (in the second loop). Where possible, for operations like these, always use the length attribute to control the bounds of your loop.

**Try This:** Write commands to initialize an array to all 0's.

**Try This:** Write commands to initialize an array with the sequence: 1, 1, 2, 3, 5, 8, 13, ...

**Try This:** Create an array called `counts` to store the number of rolls of each of the six faces of the dice using the events generated in the outcome array above.

To ensure that you did all the above correctly, write complete Processing programs to initialize and print the resulting arrays.

Several array-based computations require sequential processing of all of an array's elements. There is a special form of the for-loop that is provided. It is called the for-each loop and has the following syntax:

```java
for (variable : arrayName) {
    do something with the value of variable
}
```

That is, `variable` takes on successive values stored in `arrayName`, from the first to the last, so that it can be used in a computation in the loop.
Example: A Simple Bar Graph

As an example, the sketch below, plots a rudimentary bar graph of the energy consumption data we have seen above:

// Sketch energyBarV0, Chapter 4
String[] energySource = {"Petroleum", "Coal", "Natural Gas", "Nuclear", "Renewable", "Hydropower"};
float[] consumption = {40.0, 23.0, 22.0, 8.0, 4.0, 3.0};
void setup() {
  size(400, 400);
  smooth();
} // setup()
void draw() {
  // set up plot dimensions
  // relative to screen size
  float x = width*0.1;
  float y = height*0.9;
  float delta = width*0.8/consumption.length;
  float w = delta*0.8;
  background(255);
  for (float value : consumption) {
    // draw the bar for value
    // first compute the height of the bar
    // relative to sketch window
    float h = map(value, 0, 100, 0, height);
    fill(0);
    rect(x, y-h, w, h);
    x = x + delta;
  }
} // draw()

We should point out here that the for-each loop is useful only in situations where the loop is written purely for the sake of accessing all the values in a given array. In general, the for-loop is perhaps the most general and useful form of writing such iterations.

Array Operations

Several useful array operations are provided in Processing. These include operations for printing, sorting, and computing the minimum and maximum values stored in an array. For the examples below, assume the following two array definitions:
String[] energySource = {"Petroleum", "Coal", "Natural Gas", "Nuclear", "Renewable", "Hydropower"};
float[] consumption = {40.0, 23.0, 22.0, 8.0, 4.0, 3.0};

Printing

Occasionally, it is useful to print out the contents of an array (or a variable) in the Console Window (the one in the bottom part of your IDE where error messages appear). This can be done with the println() command:

println(consumption.length);
println(consumption);

will print the number of elements followed by the contents of the array, consumption as shown below:

6
[0] 40.0
[1] 23.0
[2] 22.0
[3] 8.0
[4] 4.0
[5] 3.0

Similarly, for printing energySource:

println(energySource);

[0] "Petroleum"
[1] "Coal"
[2] "Natural Gas"
[3] "Nuclear"
[4] "Renewable"
[5] "Hydropower"

Try This: Write commands to print out the values from energySource and consumption in the format shown below:

Petroleum, 40.0
Coal, 23.0
Natural Gas, 22.0
Nuclear, 8.0
Renewable, 4.0
Hydropower, 3.0
Min, Max, and Sorting

Often, as we will see in the example below, it is required to quickly find out the minimum and the maximum values stored in an array, or even sort or rearrange all the elements in an array in ascending/descending order. Processing provides a small handful of useful functions to do these tasks on arrays. We illustrate them below with examples:

```java
float smallest = min(consumption);
float largest = max(consumption);
```

The variable smallest will receive the value 3.0 and largest will receive 40.0. The functions `min()` and `max()` work only on arrays of `int` and `float` values. However, any array of `int`, `float`, or `String` values can be sorted in ascending order using the `sort()` function:

```java
println(sort(consumption));
```

```
[0] 3.0
[1] 4.0
[2] 8.0
[3] 22.0
[4] 23.0
[5] 40.0
```

```java
println(sort(energySource));
```

```
[0] "Coal"
[1] "Hydropower"
[2] "Natural Gas"
[3] "Nuclear"
[4] "Petroleum"
[5] "Renewable"
```

In addition to the above, Processing provides other array operations as well: reverse the ordering of elements in an array (`reverse()`), expanding the size of the array (`append()`, `expand()`), shortening it (`shorten()`), concatenating or splitting arrays (`concat()`, `subset()`, `splice()`), and copying the contents of an array (`arrayCopy()`), etc. Feel free to peek at the Processing Reference for details of these. We will introduce them in this course only if, and when, we need to use them.

Primitive and Reference Types

Now that we are familiar with arrays, it is time to get a little more intimate with some important underlying details. In Processing there are two categories of variables: **Primitive** and **Reference**. Primitive variables hold values of simple or primitive types (like `int`, `float`, etc.) and reference types are used for values that are aggregates (such as arrays, `color`, `PImage`, etc.). The distinction has to do with underlying memory models used in implementing the types and these manifest in important differences for the programmer. It is therefore crucial to clearly understand this.
In Processing, the only primitive types are **int**, **long**, **short**, **byte**, **float**, **double**, **char**, and **boolean**. When you define a variable of a primitive type in Processing:

```java
int meaningOfLife = 42;
```

You are creating a set of associations, called bindings: the variable name (**meaningOfLife**) will take on values of type (**int**) and will currently associate the value **42** with it. After the definition above, any use of the variable **meaningOfLife** will be associated with the value 42, until the program assigns a different value to it. During the lifetime of most variables in a running program the values associated with variables change and this is essentially what brings about the fundamental process of computing. Now, think about how, inside a computer the value associated with a variable is managed. For primitive types the value is stored in a designated memory location. The command above results in allocating a designated place in the computer's memory that will be capable of holding all values that are possible in the type **int**. Variables of type **int** in Processing can take on values in the range -2,147,483,648 to 2,147,483,647 and hence require 4 bytes of computer memory to store an int value. The picture in Figure 4 is what you should keep in mind when defining such variables:

The three bindings are clearly shown: the name of the variable on the left, its type on the right, and the memory 'cell' containing the current value inside the box in the middle. If commands in your program change the value of **meaningOfLife**, the value sitting inside the box changes. Any use of the value of the variable picks up the value from this cell. At the machine level the cell itself requires 4 bytes of consecutive memory locations to store the 2's complement representation of the value. But you can imagine that each of the 4 bytes that make up the cell has a unique address in the memory. We refer to the first address of the first byte of the cell as the address of the entire cell. Thus, each time a variable is used in a program, the address of the cell is generated to access the contents of the cell. The address of the cell is also called a reference.

The picture for the internal representation of arrays is different. Consider the definition below:

```java
float[] consumption = {40.0, 23.0, 22.0, 8.0, 4.0, 3.0};
```

To understand the representation of arrays, it is perhaps best to split it into the following:

```java
float[] consumption;
```

From what we have just learned, we are creating a binding of the name consumption to be an array of type float. Processing interprets this, at the lower level as shown in Figure 5:

So far this looks almost like the primitive variable representations: there is a name, a cell, and a type. However, the type is no longer **float**, but a reference to **float**. And the cell contains something called **null**. A reference, as we saw earlier, is an address of a cell in the memory. In this case, we do not yet know where and how many values will be stored in the array. For this and some other reasons, the bindings of array variables is different from those of primitive types: the name, the type of elements in the array, and the cell containing the starting address of the block of cells that store values of an array. If the block has not yet been created, as shown above, a reference has a value null. Once you create an array, as in

```java
consumption = new float[6];
```
a consecutive block of six cells, each capable of storing a float value (which also require 4 bytes each) is allocated in memory. The starting address of the first cell (i.e. the one that becomes `consumption[0]`) is stored in the cell containing the reference to float. This is shown in Figure 7 after the consumption array is filled with the desired values.

![Diagram of memory allocation]

### Processing

In Processing, variables that denote arrays and objects (which we will see in the next chapter) are called *reference variables* (or *reference types*). Other reference types you have seen before include `String`, `color`, and `PImage`. Their names are bound to cells containing a reference rather than the actual cells containing the values. This may seem confusing at first but is useful in many respects. For instance, a definition of an array, without creating one, is useful in defining an array that can then be associated with values of any size. However, you now must be a little careful when writing programs with primitive versus reference types. First, consider the following for primitive types:

```java
int x = 10;
int y;
y = x;
```

Given that both `x` and `y` are primitive types, after the assignment the cells associated with both `x` and `y` will contain the value 10. That is, an assignment between primitive types creates a copy of the value to store in the variable on the left-hand side. Next consider what happens in the case of arrays:

```java
int[] a = {10, 20, 30};
int[] b;
b = a;
```

What do you think happens? Given that both `a` and `b` are reference types, Processing defines the rule that in an assignment, what gets copied is the value of the reference cell. Thus, after the assignment, both `b` and `a` will refer to the same set of cells containing the values `{10, 20, 30}`. You can test this by adding the following lines after the commands above:

```java
b[0] = 100;
println(a[0]);
```

Before you peak ahead stop, and think. Make a picture and then write down the value that will be printed. If you wrote down 100, you are correct!

**Try This:** Make sure you run the above example to understand what is going on here.

In the parlance of those computer scientists that make it their business to design programming languages, when two variable names refer to the same data object it is called *aliasing*. Aliasing is a language trap that you must watch out for as it is fairly easy to create confusing situations like above. Aliasing, which is sometimes necessary, can lead to such unintended *side effects*. 
Arrays as Parameters

As we have seen in earlier chapters, parameterization using functions is a powerful concept in computing. Let us learn how to pass arrays as parameters to functions. We will define a function called `barGraph()` that will plot the data in a bar graph. We will use the same energy consumption data as above. The modified program from above that uses the `barGraph()` function is shown below:

```java
String[] energySource = {"Petroleum", "Coal", "Natural Gas", "Nuclear", "Renewable", "Hydropower"};
float[] consumption = {40.0, 23.0, 22.0, 8.0, 4.0, 3.0};

void setup() {
    size(400, 400);
    smooth();
} // setup()

draw() {
    background(255);
    barGraph();
} // draw()

void barGraph() {
    // set up dimensions relative to screen size
    float x = width*0.1;
    float y = height*0.9;
    float delta = width*0.8/consumption.length;
    float w = delta*0.8;

    for (int i=0; i < consumption.length; i++) {
        // draw the bar for ith data value
        // first compute the height of the bar relative to sketch window
        float h = map(consumption[i], 0, 100, 0, height);
        fill(0);
        rect(x, y-h, w, h);
        x = x + delta;
    } // barGraph()
}
```

While the sketch above performs just like the version before, it uses the `barGraph()` function to draw the bar graph. However, the `barGraph()` function is still specific to the data in the consumption array. In order to generalize this, we can parameterize the `barGraph()` function so that it can draw a bar graph for any given data set (as long as it is an array of float values). This is shown below:

```java
void draw() {
    background(255);
    barGraph(consumption);
} // draw()
```
void barGraph(float[] data) {
    // set up plot dimensions relative to screen size
    float x = width*0.1;
    float y = height*0.9;
    float delta = width*0.8/data.length;
    float w = delta*0.8;

    for (float i : data) {
        // draw the bar for ith data value
        // first compute the height of the bar relative to sketch window
        float h = map(i, 0, 100, 0, height);
        fill(0);
        rect(x, y-h, w, h);
        x = x + delta;
    }
} // barGraph()

By creating a formal parameter data, we have ‘generalized’ the barGraph() function to accept any array of float values to be plotted as a bar graph.

Try This: In the sketch above, define another data set using an array of float values. Use the barGraph() function to plot it.

Passing an array as an argument to a function is like the way simple variables are passed. Do pay special attention to the function header and notice how an array parameter is defined. One advantage of defining functions that do specialized tasks, as discussed earlier in the course, is modularization of your program. Also, notice that the barGraph() function is independent of the size of the actual data array that is provided. We could use it to plot a bar graph of any size data. This is a direct benefit from having the length attribute. It enables us to write the

![Figure 5: Visualizing USPS First Class Mail Volume, 1900-2011.](image_url)
barGraph() function without regards to the size of the array whose data is to be visualized. Additionally, if you wanted, instead, to visualize the data in the form of a pie chart, you can define a similar function called pieChart() to do so (see exercises).

Time Series Visualization

The data for energy consumption was small enough so we could easily use the array initialization feature in our program. In most cases where data visualization is involved, the amount of data to be visualized is typically too large to use array initialization. In that case it is typically stored in a data file. In many real time applications, the data might only be available over a network service, or through a sensing device. Processing provides several means for accessing data from sources whether it is files or another source. In this section, we will learn how to input data from files for doing our visualization.

Long before there was e-mail, all correspondence typically used postal services to send mail. Perhaps some of you have used a postal service for this purpose, especially for receiving goods ordered online. Figure 8 shows a snapshot of the number of pieces of First Class Mail (in billions) shipped by the United States Postal Service each year from 1950 through 2011. This type of data, which varies in discrete time steps (each year, in this case), is called a discrete time series. Such a dataset is characterized by values taken from specific points in time, or time blocks (a year in this case), and there is a finite number of possible values. For an example of continuous data, think of the current temperature in each place. It can be taken at any time and is constantly changing. As you can see, visualizing the data can immediately reveal relevant patterns. In this case we can clearly see how the volume of mail steadily increased from 1950 to 2001 and then reduced by nearly 25% in 2011 from its peak in 2001. The drastic drop in volume since 2001 can be directly attributed to increase in internet usage and e-mail in the United States over that time.

The first step in creating data visualization is to get access to the data. Not only should you ensure that the source is credible (and make sure to cite it), you also need to have it available in a form readable by your program. These days, there are several rich sources of data that you can gain access to through a simple web search. For example, do a search for “USPS first class mail volume” and you will be able to find the data source for the chart shown in Figure 4. Also, try and locate the weather data (daily temperature and precipitation log) of your town, city, etc. Once you have identified a credible data source, you have a few options to acquire the data. The data may be freely available or you may have to pay a fee to obtain it. Sometimes, a licensing fee may even be required.

Once you are past the access and usage rights, you have to determine how the data is available to you. Do you have to obtain it and store it on your computer, or can you access it online in real time? In what follows, we will assume that the data is accessible in the form of a file and we have rights of use. As an example, we will download and use historical daily stock price data for Apple Inc. for the year 2010. You can easily locate this data online. For example, we were able to get the data from the Yahoo! Finance site (finance.yahoo.com). You can select any company’s stock (for our example we picked AAPL the ticker symbol for Apple Inc.), the range you desire (we chose the entire 2010 year), and then the format you wish to receive the data. We chose the Microsoft Excel format. This option directly loads the data into rows and columns into an Excel spreadsheet which we can then save in different formats. One of the formats available is a CSV format. CSV stands for Comma Separated Values. This is a plain text format in which each rows of data is stored on a single line with column entries separated by commas (hence the name CSV). A small snapshot of the data we obtained is shown below:

```
1,4,2010,213.43,214.50,214.01,17633200
1,5,2010,214.60,215.59,214.38,21496600
```
We received 252 lines of data, one for each trading day of 2010, where each line contained the following:

MONTH, DAY, YEAR, LO, HI, CLOSE, VOLUME

where MONTH, DAY, YEAR represents the date. Thus 1,4,2010 on the first line indicates data for January 4, 2010. The three numbers that follow are the low, high, and the closing stock price. Thus, on December 31, 2010, Apple Inc.’s stock closed at a price of $322.56 trading during the day in the range $322.95 to $323.48 with 6.911 million shares trading hands.

Let us use the data above to learn how to create a visualization of a time series. To start simple, we will focus only on the closing stock price (the sixth value in each line). To begin, you should create a new Processing sketch. Let’s call it, APPL2010. Enter the following code:

```pseudocode
// File: AAPL2010V0.pde
// Sketch: Visualizing Time Series (AAPL Stock prices)
float[] price;
float X1, Y1, X2, Y2;

void setup() {
    // drawing setup
    size(600, 400);
    X1 = 50; Y1 = 50;
    X2 = width - 50;
    Y2 = height - Y1;
    smooth();
} // setup()

void draw() {
    background(0);
    // draw plot bounding box
    rectMode(CORNERS);
    noStroke();
    fill(255);
    rect(X1, Y1, X2, Y2);
} // draw()
```
When you run this sketch, you will see a dark background with a white rectangle inset. This is where we will draw the plot. So far, this sketch defines the plot area and not much else. Next, take the data file and place it in the Data folder of your sketch. All data files required by your sketch should be stored in the Data folder (just like the images). In the sketch above, we also defined an array called price (of float values) that we will use for storing the daily stock prices. At this point, we do not know how large the data set is since we haven’t yet read it. Let’s do that next.

```pde
// File: AAPL2010V1.pde
// Sketch: Visualizing Time Series (AAPL Stock prices
float[] price;
float minPrice, maxPrice;

float X1, Y1, X2, Y2;
void setup() {
  // drawing setup
  size(600, 400);
  X1 = 50; Y1 = 50;
  X2 = width - 50;
  Y2 = height - Y1;
  smooth();
  // Read the data file...
  String[] lines = loadStrings("AAPLStock.txt");

  // How long is the dataset
  price = new float[lines.length];

  // Parse the needed data
  for (int i=0; i<lines.length; i++) {
    // First split each line using commas as separator
    String[] pieces = split(lines[i], ",");
    // get the closing price of stock
    price[i] = float(pieces[5]);
  }
  println("Data Loaded: "+price.length+" entries.");

  // determine min and max stock price for the year
  minPrice = min(price);
  maxPrice = max(price);
  println("Min: "+minPrice);
  println("Max: "+maxPrice);
} // setup()
```
Most of the additions are in the `setup()` function. There are no changes in the `draw()` function that is not shown here. In `setup()` we added commands to input and parse the data to store the extracted closing stock prices in the array `price[]`. After that, we computed and printed out the minimum and maximum stock prices. When you run the sketch above, you will see the same output window as before (with an empty plot area) and you will also see the following lines in the Console Window:

Data Loaded: 252 entries.
Min: 192.05
Max: 325.47

This indicates that we were able to successfully input and parse the data and were also able to compute the stock price range for the year. Quite a year for the stock isn’t it? Even simple computations are staring pay off. Notice how the `loadStrings()` function is used to input the entire file as an array of string. Each line in the file gets stored in the array `lines[]` as a separate string. Each string is then parsed by first breaking it into pieces using the `split()` string function resulting in an array pieces where each data item on the line is stored as a string. Next, we convert the needed data item, the closing stock price (at `price[5]`) into a float value and store it in the `price[]` array. Now that we have our data where we need it, in the `price[]` array, we are ready to do the next step: visualize.

```java
void draw() {
    background(0);

    // draw plot bounding box
    rectMode(CORNERS);
    noStroke();
    fill(255);
    rect(X1, Y1, X2, Y2);

    drawGraph(price, minPrice, maxPrice);
}
```

```java
void drawGraph(float[] data, float yMin, float yMax) {
    stroke(0);

    beginShape();
    for (int i=0; i < data.length; i++) {
        float x = map(i, 0, data.length-1, X1, X2);
        float y = map(data[i], yMin, yMax, Y2, Y1);
        vertex(x, y);
    }
    endShape();
}
```

The sketch now plots the graph shown below. The `drawGraph()` function does all the work. Given the `data[]` array, the minimum and maximum values in it, and the bounds of the plotting area `(X1, Y1)` and `(X2, Y2)` it is able to plot the daily stock price. It uses the `map()` function to map each data value to the proper `x` and `y` coordinates within the plot area. Carefully study the
parameters supplied to the map() function and make sure you understand them. We are using the beginShape() and endShape() functions to specify each data point as a vertex in the plot.

**Try This:** Now you can start experimenting with different aspects of the sketch. For example, vary the size of the sketch itself (from 600x400 to some other values). Does the plot scale appropriately? Why?

Figure 6: Drawing the graph of APPL Stock using drawGraph()

**Try This:** Change the stroke color of the graph to red or another color of your choosing.

**Try This:** Change the vertex() command to the curveVertex() command to draw a smooth curve. You will additionally need to define the start and end anchor points.

**Try This:** Turn this graph into an area plot. An area plot shows the entire area under the graph in a different color. HINT: You will need to experiment with optional parameters of beginShape() or endShape().

It is time to further refine the graph. This time we will add the legend, and label axes. For this, you will need to create a font by adding the following at the top of the sketch (above setup()):

```plaintext
PFont legendFont = createFont("SansSerif", 20);
```

And then add this command in drawing setup section of setup():

```plaintext
textFont(legendFont);
```

Next, we show the new version of draw() function:

```plaintext
void draw() {
  background(0);
}
```
// draw plot bounding box
rectMode(CORNERS);
noStroke();
fill(255);
rect(X1, Y1, X2, Y2);

drawGraph(price, minPrice, maxPrice);

// draw legend
// title
fill(255);
textSize(18);
textAlign(LEFT);
text("(AAPL) Apple Inc. 2010", X1, Y1 - 10);
textSize(10);
textAlign(RIGHT, BOTTOM);
text("Source: Yahoo! Finance (finance.yahoo.com)", width-10, height-10);

// axis labels
drawXLabels();
drawYLabels();
} // draw()
Simple Data Modeling

We will conclude this section with a small yet effective computational enhancement to the stock price visualization. First, let us consider the task of computing the average stock price of Apple Inc. stock for the data above. We can write a function to compute this as follows:

```java
float average(float[] data) {
    float sum = 0;
    for (value : data)
        sum += value;
    return sum / data.length;
} // average()
```

The function above accumulates all the stock prices into the variable `sum` and then returns the average by dividing `sum` by the number of elements it accumulated. We could use this function in our sketch by using the command:

```java
float avStockPrice = average(price);
```

While an average stock price is a useful indicator, in the financial world, it is more useful to compute the average stock price over a smaller period. For example, we may be interested in viewing the price over a 10-day trading range. Computing the 10-day average across the entire...
year is a useful way to model the movement of a stock. This is called a *simple moving average* and is considered a better indicator of the average price of a stock over a given time period. Over time, this moving average can be plotted along with the daily stock price as shown below:

```java
void movingAverage(float[] data, float yMin, float yMax, int MAP) {
  noFill();
  stroke(255, 0, 0);
  strokeWeight(2);
  beginShape();
  for (int i=MAP-1; i < data.length; i++) {
    float sum = 0;
    for (int k=i-(MAP-1); k <= i; k++) {
      sum += data[k];
    }
    float MA = sum/MAP;
    float x = map(i, 0, data.length-1, X1, X2);
    float y = map(MA, yMin, yMax, Y2, Y1);
    vertex(x, y);
  }
  endShape();
} //movingAverage()
```
Adding this to your sketch and calling it from the draw function will plot a red curve of the moving average. This is shown below for a 25-day moving average:

![Figure 8: Apple Inc. stock price plot with 25-day moving average.](image)

As you can see, the moving average plot is a smoother rendering of the price plot as it eliminates the daily ups and downs. In the financial world, a moving average also reflects the stock’s momentum. Whenever the stock price curve is above the moving average it implies that the stock price has an upward momentum and generally it is considered a good buy. Whenever the moving average is above the stock curve it reflects a downward momentum of the stock. Thus, in early February, and in May, July and August the Apple stock had a downward momentum. For the entire years’ worth of data though we are looking at a time in history when the stock had an upward momentum for most of the year starting from March 2010. Can you guess why? The iPad was launched on April 1, 2010.

### The Data Visualization Process

The example above illustrates how data, once stored in the array can be used to create effective plots or other kinds of visualizations. Even simple visualizations can lead to key insights that may not be obvious in a table of numbers. These days data visualization is a growing industry. Beyond business analytics, visualizations enable us to understand and represent complex phenomena. While computers and computational processes have created a richer medium for data visualization the process of visualization itself predates computers. As we learn more concepts in computing, we will be able to store more sophisticated kinds of data, process it in numerous ways, and use the expressive power of Processing to create rich visual representations. We like the following quote from Nathan Yau:

“When you think of visualization as a medium rather than a monolithic tool, it’s something much more flexible that can be used for a lot of things. It’s also more exciting. You can tell stories with
data through analysis, journalism, or art. Visualization can be fun or serious; it can be beautiful and emotional or barebones and to the point.”

The process of data visualization itself was formalized by Ben Fry in his PhD thesis where he outlined the various stages of creating data visualization: acquiring the data, parsing it, filtering it, mining it, choosing a visual representation, refining of improving the visual representation, and finally making the visualization interactive. The last, making it interactive, is where the power of computers combined with languages like Processing really shine. We will explore interactivity later in this book. Some examples of making your sketches from this chapter interactive are included in the exercises.

Despite the formalization, data visualization still relies on one’s creativity and sense of aesthetics. Also, as Nathan Yau points out, there are a number of do’s and don’ts that one must keep in mind. These include obvious, yet important things like checking the data that forms the basis of a visualization to make sure that it is free of errors and is valid. Moreover, all visualizations should be properly labeled to include the coding(s) used in the visual representations, labeling the axes, including the units, citing the proper sources, etc. Visual representations, as effective as they may be, can also create false impressions if there are flaws in the geometry of the chosen graphics.