# Session 4 - Acquiring and parsing data

This session is designed explore in greater depth the process of parsing data for visualization. The programming processes required to break down data files into components and store those relevant to the visualization is developed.

## By the end of this session you should be able to:

* extract data values from a text file using simple splitting
* use arrays in your sketches to store collections of data items
* extract data values from a text file using regular expressions
* organise your Processing sketch by splitting tasks into methods and tabs
* use Processing to to read and write JSON, XML and other file types

## Contents

1. [4.1 Introduction](http://moodle.city.ac.uk/mod/page/view.php?id=580589#s4.1)
2. [4.2 Working with text](http://moodle.city.ac.uk/mod/page/view.php?id=580589#s4.2)
3. [4.3 Handling data interchange formats](http://moodle.city.ac.uk/mod/page/view.php?id=580589#s4.3)
4. [4.4 Conclusions](http://moodle.city.ac.uk/mod/page/view.php?id=580589#s4.4)

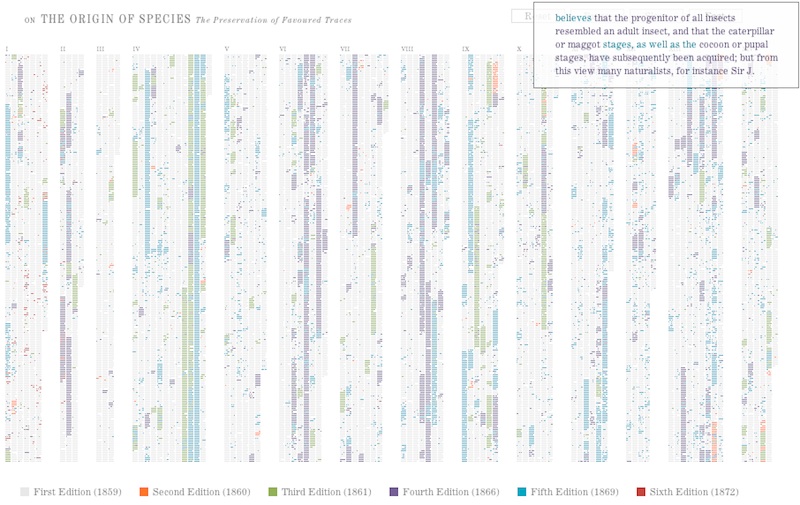
## 4.1 Introduction

We've seen in a previous session how we can read text and numbers from a table and use those data in our sketches. This session considers in greater depth how we go about *parsing* data so we can structure it in a way that is useful to us to visualize. If you recall, parsing forms the second stage of Fry's 7-stage workflow. After acquiring data it represents the process we need to go through to add structure and meaning to data before we can go about filtering, mining and representing it. 

Let's start with an example to illustrate how data parsing was used to show how Darwin's text *On the Origin of Species* itself evolved over time - [The Preservation of Favoured Traces](http://fathom.info/traces)

Written in Processing, this example of data visualization provides both an overview of Darwin's text (one column per chapter), gives us details on demand (interactively moving the mouse over the text displays the selected section), and uses colour to show where changes have occurred over time. In order to construct this visualization, it would have been necessary to *parse* each line of text in the original book(s) in order to compare it with previous editions. This allows added and removed sentences, entirely new chapters, edited words etc. to be identified.

We will examine how we go about processing text and numbers so we can parse our own datasets when constructing visualization. In order to do so, we will need to introduce a few new programming concepts such as *methods* and *arrays* as well as looking at some established approaches to parsing and structuring data such as *regular expressions* and *XML markup*.

[](http://staff.city.ac.uk/~jwo/datavis/session04/images/darwin.jpg)

Output from Ben Fry's Preservation of favoured traces.

## 4.2 Working with Text

Let's suppose you are writing a Processing sketch to visualize trending topics being sent via Twitter. Your application generates a simple file that contains the tweets associated with trending hashtags. The file [trendingTweets.txt](http://staff.city.ac.uk/~jwo/datavis/session04/textRead1/data/trendingTweets.txt) looks like this:

The Independent is dead! Self-immolation of their broadsheet image through incessant Upworthy-style content? http://ow.ly/YfCN0

The closure of The Independent and Independent on Sunday has been confirmed by owner Evgeny Lebvedev http://ow.ly/3beWsW

I fear @TinaMPurcell and I are responsible for the demise of the @Independent. We cancelled our last order (Saturday edition) 2 weeks ago.

Britain's 'The Independent' newspaper to go digital-only: Britain's The Independent newspaper will close its p... http://fniq.a.boysofts.com/bIG

The Independent to cease printing, that's one less newspaper that isn't right-wing/Tory in this country then! #IGiveUp

At least the Independent was a newspaper people bought to read while the Sun is not a newspaper just a cheap form of toilet paper! #wato

I think the #fiscalframework is more complicated, and harder to understand, than #gravitywaves

My dad (a physicist) is "ecstatic" about the #gravitywaves thing. Unprecedented in Khan language!

Everybody seems to be excited about the #gravitywaves. Now image a world where all the money we've used to save banks was put into science.

We made some waves in 2015, but the gravity wave detection is just tops! CONGRATULATIONS @LIGO TEAM! #Plutoflyby #GravityWaves #LIGO

Some of these stars in the 'Orion' constellation are closer to Earth than they are to each other. #DeepSpaceThoughts

#gravitywaves paper is very readable by just about anyone and open access. http://journals.aps.org/prl/abstract/10.1103/PhysRevLett.116.061102

We just re-confirmed #gravitywaves by throwing a banana in the air.

LISTEN: This is the sound of two black holes colliding http://on.rt.com/74aq #LIGO #gravitywaves

Most of the #gravitywaves reporting is terrible. This story developed 1.5 billion years ago! This is old news.

Why @HillaryClinton's #DemDebate performance won't do a darn thing for her campaign: http://bit.ly/1WhsXTh

Enjoying #DemDebate & agree w/ @BernieSanders that we should not use children to send a message to any1 #Not1more

Why doesn't @BernieSanders bring up the TPP when @HillaryClinton tries to imply Wall Street doesn't influence @BarackObama #DemDebate

Doing marijuana, "Big Bird" & more: The best, worst & ugliest Twitter reactions to tonight's #DemDebate http://slnm.us/CN51sHM

Once again, @HillaryClinton tries to reinvent her image for tonight's #DemDebate...

Happy Birthday to Charles Darwin! The famed naturalist was born on 2/12/1809. #DarwinDay http://trib.al/ZTjw7NK

Published around 140 years ago, #Darwin is still cited in 181 documents in Scopus from 1976 to this day. #DarwinDay

"If everyone were cast in the same mould, there would be no such thing as beauty." -- Charles Darwin #DarwinDay

"Ignorance more frequently begets confidence than does knowledge" #CharlesDarwin

Happy #DarwinDay @cdarwin to quote you"cordial good wishes for success of all kinds & may all your theories succeed"

In this example, the data are not in an obvious tabular format so we cannot use the Table class which we used previously to read tab-separated value (TSV) files. Let's instead see how we can store the content of the file in Processing variables inside a sketch. We will store all the lines of the text file in an *array* - a way of storing a collection of variables together inside a single entity. An array is declared like any other variable, except we use the [ and ] square brackets to indicate its array status. For example,

|  |
| --- |
| float[] ages; |

would declare a variable for storing not one, but a whole collection numbers. We can say how many numbers are stored inside the array variable by initialising it with the new command. For example,

|  |
| --- |
| ages =new float[10]; |

would allow 10 different float values to be stored in a single array variable. To access an individual value in the array, we can use an index value inside square brackets, where the first value in the array is index 0, the second value is index 1 etc. For example:

|  |
| --- |
| ages[0] = 31;  ages[1] = 27;  ages[2] = 42;  ages[3] = 35; |

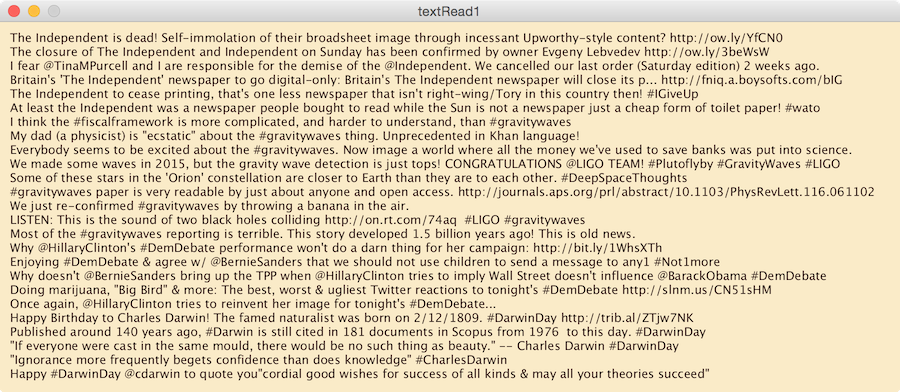
In practice we often use Processing methods to make the task of filling arrays with data and extracting values from arrays much simpler. Consider the following example that displays each of the tweets in the trendingTweets.txt file

|  |
| --- |
| // Reads and displays some simple text in a sketch  // Jo Wood, 12th February 2016    String[] lines;   // Stores all the lines of text to display.    void setup()  {    size(900,370);      fill(40,15,15);         // Dark brown text    textSize(12);           // 12pt text size.      // Read the lines of text from a file into a String array    lines =loadStrings("trendingTweets.txt");  }    void draw()  {    background(250,235,200);    float yPos = 20;   // Position of text.      for (String textLine : lines)    {      text(textLine,10,yPos);      yPos = yPos + 14;    }      noLoop();   // Display only once.  }} |

The first line declares an array of Strings each element of which is capable of storing some text. In this example, each element will store a single Tweet. We can transfer the text from trendingTweets.txt into our array with the Processing command [loadStrings()](http://processing.org/reference/loadStrings_.html). In this case, the loadStrings() command will create the array of the correct length for us (depending on the number of lines in the file being read), so we don't need to set its size explicitly with the new command. As reading the text file is a one-off operation, we place it inside the setup() method. The draw() method then loops through each String in the array and displays the text in the sketch. It uses a special simplified version of the for loop known as the 'for each' loop. So the line

|  |
| --- |
| for (String textLine : lines) |

should be read as 'for each string in the array lines, store the text inside a new variable called textLine'. We let Processing do the counting of the lines so we don't need to create and update any counting variable ourselves. Inside the loop, we get Processing to display the contents of textLine (which will change to each new line of text in the file every time we repeat the code inside the for-each loop). We use the float variable yPos to keep track of the vertical position at which we need to display the text, and move it down by 14 pixels every time the loop is repeated.

[](http://staff.city.ac.uk/~jwo/datavis/session04/images/tweet1.png)

Output from textReading sketch *(click to enlarge)*.

The parsing involved in this sketch is very simple - we are simply splitting a text file into individual lines before we display them. In most data visualization sketches you write, you will probably want to do something a little more sophisticated with the data you are parsing. A common task when reading text is to split it into individual words or *tokens*. We can do this using Processing's [split()](http://processing.org/reference/split_.html) command to split some text into tokens. Given a string containing a collection of characters, split() will create an array of new strings by splitting up the text using some *delimiter* text. For example, to split the sentence "*The cat sat on the mat.*" into individual words we could use the line:

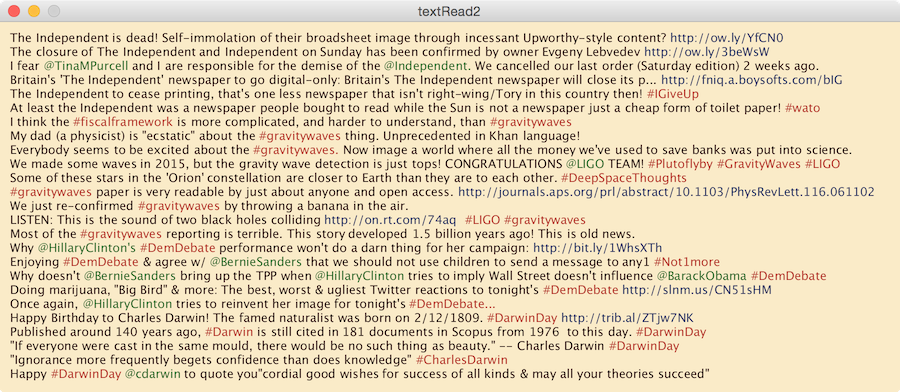
|  |
| --- |
| String[] words =split("The cat sat on the mat."," "); |

By splitting the text using the space character as a delimiter the result would be a string array where the first element (words[0]) would be "The", the second element (words[1]) would be "cat" etc. Note that whenever you refer to the contents of a string (as opposed to the name of a String variable), you use the " symbol around the beginning and end of its contents.

So, to enhance our tweet parsing sketch to highlight #hashtags, @usernames and URLs, we could do the following:

|  |
| --- |
| // Reads and displays some simple text and hashtags in a sketch  // Jo Wood, 12th February 2016    String[] lines;   // Stores all the lines of text to display.    void setup()  {    size(900,370);    textSize(12);           // 12pt text size.      // Read the lines of text from a file into a String array    lines =loadStrings("trendingTweets.txt");  }    void draw()  {    background(250,235,200);    float yPos = 20;   // Vertical position of text.      for (String textLine : lines)    {      // Split each line into words      String[] words =split(textLine," ");      float xPos = 10; // Horizontal position of text.        for (String word : words)      {        if (word.startsWith("#"))        {          fill(183,56,46);   // Red        }        else if (word.startsWith("@"))        {          fill(51,108,47);   // Green        }        else if (word.startsWith("http"))        {          fill(46,56,103);   // Blue        }        else        {          fill(40,15,15);   // Dark brown        }          text(word+" ",xPos,yPos);        xPos = xPos +textWidth(word+" ");      }      yPos = yPos + 14;    }      noLoop();   // Display only once.  } |

The draw() method loops through every line in the text file and then splits each line into words, storing the result in the String array words. The sketch then uses a second, nested, 'for each' loop to go through each word in the line looking to see if it starts with the character '#' or '@' or the text 'http'. It does this by using the method startsWith() which can test to see if a given string starts with a given character or characters. If the word does start with one of these, the text fill colour is set to red, blue or green, if not, it sets it to brown. Finally within this second loop, we need to keep track of the pixel position of the last word drawn, so we do this with the variable xPos using Processing's [textWidth()](http://processing.org/reference/textWidth_.html) method for working out how many pixels wide a given bit of text is.

[](http://staff.city.ac.uk/~jwo/datavis/session04/images/tweet2.png)

Output from enhanced text reading sketch*(click to enlarge)*.

### 4.2.1 Case Study: Parsing UK GDP data

The ability to read a text file, split it into lines and tokens and then look for certain letters or phrases is the core of much text parsing. Let's consider a more typical data visualization task that parses a more complex data file. The following table shows the first few lines of a dataset representing UK Gross Domestic Product (GDP), taken from the [Guardian Data Blog](http://www.guardian.co.uk/news/datablog/2009/nov/25/gdp-uk-1948-growth-economy). When downloading from the Guardian Data Blog site, the data are usually provided as a [Google Spreadsheet](http://www.google.com/google-d-s/spreadsheets/):

Data can be exported from the spreadsheet by selecting 'Download as text' from the 'File' menu, allowing you to save the table in .tsv format. In this example, data are stored in a file [gdp.tsv](http://moodle.city.ac.uk/mod/page/gdpSketch/data/gdp.tsv). To make reading the data simpler, it is worth editing the text file and adding a # symbol to the first two header lines so we can distinguish column titles from the data themselves. After editing, the first few lines look like this:

# IHYQ ABMI YBHA IHXT IHXW

#Gross Domestic Product in Year, quarter Gross Domestic Product: Quarter on Quarter Growth. Chained volume measures: Seasonally adjusted. GDP, not adjusted for inflation £m Non-inflation Chained volume measures: Seasonally adjusted

1955 Q1 85,937 4,749 93 1,688

1955 Q2 0.2 86,091 4,773 94 1,690

1955 Q3 2.1 87,877 4,941 97 1,723

1955 Q4 -0.6 87,384 5,027 98 1,711

1956 Q1 0.7 87,990 5,124 100 1,721

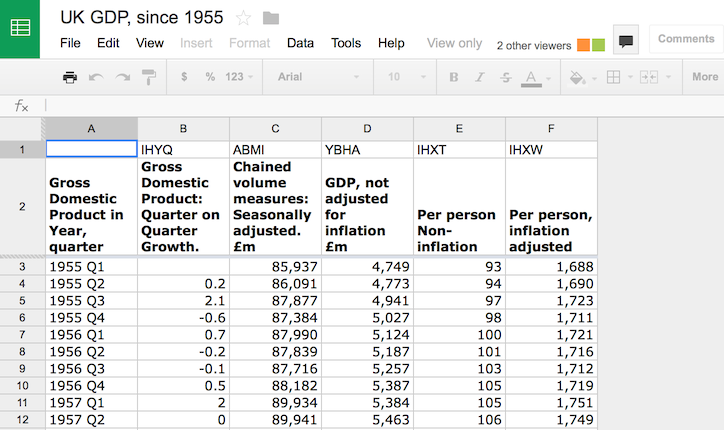
1956 Q2 -0.2 87,839 5,187 101 1,716

1956 Q3 -0.1 87,716 5,257 103 1,712

1956 Q4 0.5 88,182 5,387 105 1,719

## Edit data or edit Processing code?

In many cases the data you wish to use in your sketches will not arranged in the ideal format to make parsing easy. You are faced with a choice - do you edit the data file directly to make it easier to parse, or do you write code in Processing to deal with the particular way the data are arranged? Edits to data might include adding # symbols to lines you wish to ignore, or deleting them entirely. In the gdp.tsv example above, I added #s to the first two lines before doing any programming with Processing. This made sense because it was a quick 'one-off' thing to do that saved considerable Programming time later on. For more sophisticated editing tasks you might consider using some kind of ***data wrangling software*** such as the commercial package (but free) [Trifacta](https://www.trifacta.com/) or the opensource [Open Refine](https://github.com/OpenRefine/OpenRefine/wiki).   
  
Other changes might be more tedious to edit, such as replacing the blank entries in a table with 0s or reformatting dates. In which case, writing Processing code to account for the way the data are arranged can be quicker and more flexible (e.g. allowing other datasets in a similar format to be substituted without any additional editing).   
  
Which approach you take will depend on the complexity and size of the data you are dealing with, how confident you feel with programming and how flexible your sketch needs to be. Be pragmatic - take the options that involve the least work.

[](http://staff.city.ac.uk/~jwo/datavis/session04/images/googleSpreadsheet.png)

Google GDP spreadsheet *(click to enlarge)*.

For the purposes of this example, let's assume we wish to visualize inflation-adjusted GDP per person over time (column F of the original spreadsheet) in order to answer the question *How does the severity and duration of the last economic downturn compare with previous recessions?* We could use Processing's loadTable() to read this table of data, but here we shall use our own data parsing in order to understand how data are transferred into Processing sketches.

The file contains tab-separated values defining the columns of data, but there are a few problems that we will have to deal with when reading individual items. Notice how the GDP monetary values contain commas within them and dates are referred to with year numbers and quarter numbers. The commas in the monetary values represent thousand place holders (e.g. to make the number 1000000 easier to read as 1,000,000). They are not decimal point symbols as used in much of continental Europe, nor are they column separators as they would be in a comma separated (CSV) file.

We will build our sketch to visualize the GDP data in stages. This can be a useful approach to take when tackling a more complex data visualization task. Our first stage is to create a basic sketch structure and check our sketch can read the data values from a file. The code below shows this first stage.

|  |
| --- |
| // Reads and displays GDP data.  // Jo Wood, 12th February 2016    float[] years;   // Stores a collection of dates.  float[] gdps;    // Stores a collection of GDP values.    void setup()  {    size(400,300);    readData();  }    void draw()  {    background(255);      noLoop();      // Draw once only.  }    // Reads the GDP data from a text file.  void readData()  {      String[] textLines =loadStrings("gdp.tsv");        for (String  textLine: textLines)      {        println(textLine);      }  } |

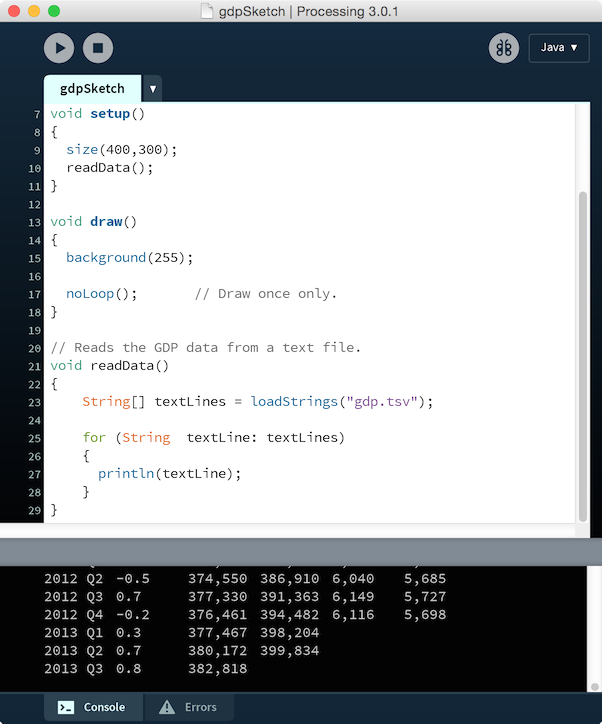
It contains the normal structure containing setup() and draw() methods. In this case we do no drawing other than create a white background and tell Processing not to repeat the drawing at 60 frames per second. We will later be drawing data items that will be stored in the arrays years and gdps, so these are declared at the top of the sketch.

The setup() method will need to read the data in before anything can be drawn, so here we call a new method which we have named readData(). In contrast to Processing's pre-defined methods like draw() and setup(), this is an example of creating our own method to perform a task that we choose to define. User-created methods like this should be named just like variables, starting with lower case letters but ending with () brackets. It is important to be able to distinguish *calling a method* as we do with the line readData(); in setup() and *defining a method* as we do in the last 9 lines of this sketch.

The definition of our own readData() method in the last 9 lines simply uses Processing's [loadStrings()](http://www.processing.org/reference/loadStrings_.html) to read the text file into an array of Strings called textLines. We then use a 'for-each' loop to iterate through each String in the array, and in this first instance, simply display the text on screen. This uses a helpful command [println()](http://www.processing.org/reference/println_.html) for testing and developing a sketch. The command (pronounced 'print line'), displays the contents of any variables or text placed inside the brackets. This will appear as white text in the black console at the bottom of the Processing window, not in the sketch itself.

The loop in readData should be enough to check that our sketch has found the TSV file and the lines of text are what we expect them to be.

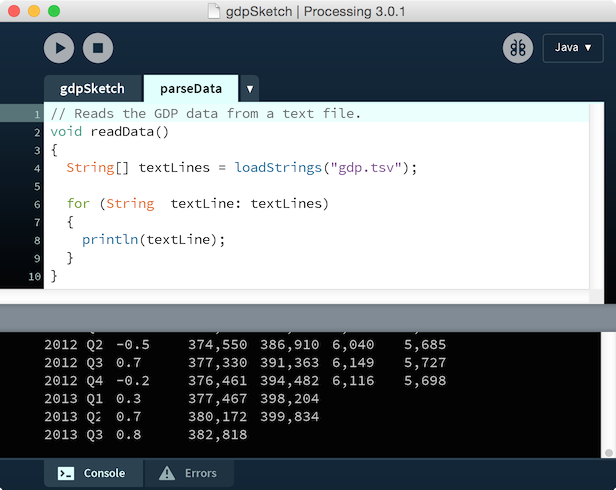
Before reading on you may wish create the sketch shown above by copying the code and dragging [gdp.tsv](http://moodle.city.ac.uk/mod/page/gdpSketch/data/gdp.tsv) into it. Check that it produces some output in the black Processing console similar to that shown below:

[](http://staff.city.ac.uk/~jwo/datavis/session04/images/processingConsole.png)

Now that we have the beginnings of a working sketch, let's add code to parse the data from the text file. As your sketches become more complex, it is useful to be able to group together the methods that perform similar tasks and separate those from methods that do other things. Processing helps you to do this by creating new *tabs* each related to different tasks. You can think of a tab as a folder into which a block of code is placed to help you keep a more complex sketch organised.

In this case we will create a new tab called *parseData* where all the code for parsing the text file will be placed. To create a new tab, click the right-arrow button towards the top-right of the Processing window and select New Tab.

Create a new parseData tab in the GDP sketch and then cut the code from the user-defined method readData() and paste it into the new tab. Your Processing editor should now look something like this:

[](http://staff.city.ac.uk/~jwo/datavis/session04/images/processingWindowTab.png)

Now we have a convenient place in which to put our parsing code, let's consider how we can improve it to read in two columns from the GDP file.

|  |
| --- |
| // Reads the GDP data from a text file.  void readData()  {    String[] textLines =loadStrings("gdp.tsv");      // Count the number of data rows in the file.    int numRows = 0;    for (String textLine : textLines)    {      if (!textLine.startsWith("#"))      {        numRows++;      }    }    years =new float[numRows];    gdps  =new float[numRows];      // Read the data into the arrays.    int dataRow = 0;    for (String textLine : textLines)    {      if (!textLine.startsWith("#"))      {        String[] tokens =split(textLine,TAB);          // Read the year data from the first 4 characters in the first column        float date =float(tokens[0].substring(0, 4));          // Read quarter figure from end of the first column and add it to year.        date = date +float(tokens[0].substring(6))/4 - 0.25;          // Store date in the array.        years[dataRow] = date;          // Read the inflation-adjusted GDP per person value from the 6th column.        String gdpText = tokens[5].replace(",","");       // Remove the commas.          if (gdpText.length() == 0)                 // Replace blank cells with 0s.        {          gdpText ="0";        }        gdps[dataRow] =float(gdpText);     // Convert text to a number and store.          // Update the current data row.        dataRow++;      }    }  } |

The first thing this method does is read in the entire file into an array of strings and then loop through each line counting up the number of rows of data. This is pretty much as our first version of readData() except this time we add one to a counting variable numRows only if the line does not start with the # character (note the ! symbol at the start of the ifline indicating 'not' - if the text line does not start with a '#', then do the stuff in the curly braces that follow).

After looping though each of the textLines text array elements, we now know the number of data rows in the table so we can initialise the two new sketch arrays years and gdps (declared at the start of the main sketch tab) to hold the relevant number of items.

The remains of the method reads through the array of text lines again, this time extracting the value in column 0 (the year and quarter) and column 5 (GDP per person adjusted for inflation). The array index corresponding to each row in the table is stored in the number variable dataRow.

Each row is split into the individual table cell values using the split() command, in this case splitting using the TABdelimiter and storing the results in the tokens array.

To store each year as a number we need to extract the first four digits of each line (have a look at [gdp.tsv](http://moodle.city.ac.uk/mod/page/gdpSketch/data/gdp.tsv) above to remind you why this is the case). To do this we do two things, both in the same line of code. Firstly we use [substring()](http://processing.org/reference/String_substring_.html) to extract only a limited number of characters from the string. Secondly we convert the four-character string into a number variable using the Processing command [float()](http://www.processing.org/reference/float_.html) (note the brackets after the word indicating it is a command). The results are stored in a number variable we have called date.

Our dataset includes GDP figures for every financial quarter of each year, that quarter being indicated by a Q1, Q2, Q3, or Q4, in the first column. We can extract the crucial number 1-4 by using [substring()](http://processing.org/reference/String_substring_.html) again, this time extracting the 7th character of the first column. This number is converted into its numeric equivalent by dividing by 4 and subtracting 0.25 (so, for example, '2012 Q4' becomes '2012.75').

As we found with the bad teeth datasets, sometimes your data may contain blank cells where data are unavailable. We have at examples of blank cells in 2013. We could have edited the data first and replaced these blanks with 0s, but in this case we use the line if (gdpText.length() == 0) to look for cells with no length (i.e. blank) and replace them with 0s inside our sketch.

Finally, we extract the GDP figure and convert it into a number. In this case, we need to get rid of the place-holding commas in the number, so use another string parsing command replace() that allows us to replace each instance of "," with an empty string "" (effectively removing it).

There was quite a lot of new programming commands being used in the parsing in this example. But one of the advantages of grouping behaviour into tabs in your sketch is that once it is working, we can forget about it and focus our attention on the other tabs doing the more interesting visual design part of our sketch.

So to represent the GDP data visually now that they have been read, parsed and stored in two arrays - years and gdps, the rest of our sketch focusses on the drawing of the GDP data as a line chart:

|  |
| --- |
| // Reads and displays GDP data.  // Jo Wood, 26th January 2016    float[] years;   // Stores a collection of dates.  float[] gdps;    // Stores a collection of GDP values.    float minYear, maxYear;  float minGDP  =MAX\_FLOAT;  float maxGDP  =MIN\_FLOAT;    void setup()  {    size(400, 200);    readData();    findMinMax();  }    void draw()  {    background(255);      beginShape();   // Start a line shape.    for (int i=0; i<years.length; i++)    {      if (gdps[i] > 0)      {        // Plot each coordinate pair in the line.        float x =map(years[i], minYear, maxYear, 0,width);        float y =map(gdps[i], minGDP, maxGDP,height, 0);        vertex(x, y);      }    }    endShape();   // End the line shape.      noLoop();     // Draw just once.  }    // Finds the minimum and maximum values in the data.  void findMinMax()  {    for (float gdp : gdps)    {      minGDP =min(minGDP, gdp);      maxGDP =max(maxGDP, gdp);    }    // Years are in order, so min/max are first and last elements.    minYear = years[0];    maxYear = years[years.length-1];  } |

The start of the sketch declares those variables that will be used by more than one method. These include the minimum and maximum values of the years and GDP values we will be plotting and the arrays storing the data themselves.

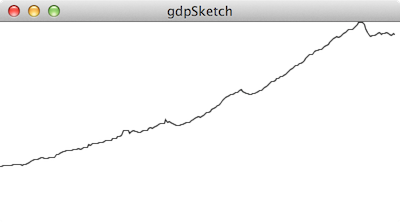
The setup() method calls the readData() method described above. We know that once it has been called, years and gdps will contain all the data we need, extracted from the text file. We then call another user-defined method findMinMax() which is itself defined towards the bottom of the sketch.

findMinMax() just loops through the arrays looking for the largest and smallest GDP values in much the same way as we did when looking at dental health data in Session 2. In this case as the data are in an array, we can use the in-built property of all arrays, length that finds out how many elements are stored in the array. The same property is used to find the last element in the years array to store the most recent year value.

When you have well-defined tasks like mining the data values for their range, it is often easier to manage if they are placed in their own method like this one. You can then re-use the method for other similar range-finding tasks.

The draw() method should look largely familiar - it draws a line between each pair of year-GDP data, each scaled to the width and height of the sketch using the [map()](http://processing.org/reference/map_.html) command. What is new is a set of Processing commands [beginShape()](http://processing.org/reference/beginShape_.html)[vertex()](http://processing.org/reference/vertex_.html) and [endShape()](http://processing.org/reference/endShape_.html), which are used for drawing more complex shapes comprising a set of coordinate pairs.

Clearly there is scope for more sophisticated graphical representation, using colour, showing graph axes and possibly including some interaction, but this session is all about parsing, so we will leave the representation for now. In its present form it is sufficient to show that the recent recession has resulted in a decline in GDP more rapid than any we have seen in the UK over the measurement period (1955 onwards), but that decline has stopped in the last two years. We have yet to see whether this there will be a GDP-related recovery will emerge, at least from these data. We might also use this as a basis for comparison with other measures of quality of life such as employment, health and crime figures.

[](http://staff.city.ac.uk/~jwo/datavis/session04/images/gdp1.png)

Simple output from GDP parsing sketch.

### 4.2.2 Summary of Text Parsing Commands

The table below gives a summary of the various Processing commands that have been used to parse the data in the examples above. This is not an exhaustive list, but it should give you a good idea of the common parsing tasks when reading data from a text file. For a more complete descriptions see the [Processing String reference](http://processing.org/reference/String.html) and the [Java String API](http://download.oracle.com/javase/6/docs/api/java/lang/String.html)

|  |  |
| --- | --- |
| **Command** | **Behaviour** |
| [length()](http://processing.org/reference/String_length_.html) | Reports the number of characters in a string. |
| [substring()](http://processing.org/reference/String_substring_.html) | Extracts part of a string. |
| [replace()](http://download.oracle.com/javase/6/docs/api/java/lang/String.html#replace(java.lang.CharSequence,%20java.lang.CharSequence)) | Replaces all instances of one bit of text with another |
| [split()](http://processing.org/reference/split_.html) | Splits a string into an array of smaller strings based on the given delimiter. |
| [trim()](http://processing.org/reference/trim_.html) | Strips any whitespace of the start or end of a string |
| [int()](http://processing.org/reference/int_.html) | Converts a string into a whole number (or NaN if cannot convert). |
| [float()](http://processing.org/reference/float_.html) | Converts a string into a decimal number (or NaN if cannot convert). |
| [nf()](http://processing.org/reference/nf_.html) | Converts a number (either an int or float) into a String. Can be used to format to a fixed number of decimal places. |

### 4.2.3 Regular Expressions

This short section on regular expressions covers some optional advanced material. If you are not familiar with regular expressions or are not feeling confident in your programming, you can skip this section. It is for background information only for those wishing to apply their previous programming experience to data parsing.

*Regular expressions* allow sophisticated parsing of text and are available in many programming languages. They are very powerful and flexible, but can look rather cryptic. Processing allows access to Java's regular expression engine to split text into tokens or to search for given patterns in text.

To use regular expressions when splitting text into tokens, use the String version of split() that is, instead of split(myString,delimiter); you use myString.split(*regex*);

For example,

|  |
| --- |
| String textLine ="item 1 : item 2: item3    : item 4 : Item number five.:item 6    ";    String[] tokens = textLine.trim().split("\\s\*:\\s\*"); |

will split the text into separate tokens using the : divider stripping out any whitespace at the beginning or end, but not middle of each token.

Note that when you use the regular expression '\' (backslash) character, it needs to be 'escaped' in Processing/Java by including double backslashes.

It is beyond the scope of this module to describe how to use regular expressions in any detail, but a useful summary of regular expression symbols can be found in the [Java Pattern API documentation](http://download.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#sum). You may also find the online [regexp editor](http://www.myregexp.com/) useful for testing regular expressions before they are coded in Processing.

We can also use Processing/Java to *match* rather than *split* text. This uses the [Java Pattern class](http://download.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html) to store the pattern matching code. Unless you are confident using regular expressions, you probably won't need to make use of this class, but it is referred to here for reference.

## 4.3 Handling data interchange formats

Some data formats do not lend themselves well to the kind of parsing we have described so far. Splitting text into lines and tokens works well for tabular data, but less well for data that are not so obviously structured as tables such as hierarchical or marked up data. Since there is often a need to handle data in this form, we will consider two commonly used flexible interchange formats used in data handling and transfer.

## 4.3.1 The JSON data format

[JSON](https://en.wikipedia.org/wiki/JSON) (and see [www.json.org](http://www.json.org/)) is an increasingly used text format for representing data objects using 'key-value' pairs. While originally associated with and often applied to the JavaScript language (for example it is commonly used with [D3 javascript visualization library](https://d3js.org/)), the specification is language neutral and works well for data that can be described by a collection of values each with some identifying name.

It is beyond the scope of this module to discuss JSON in detail, but understanding the core concepts behind its specification is helpful when trying to visualize data provided in this format.

At its heart is a set of key-value pairs providing the ability to identify any data item (value) by a unique name (key). For example, suppose we wished to represent some details about a pet rabbit, we might create the following JSON file. The open and closed curly brackets or braces are used to group these key-value pairs into an *object*:

{

"name": "Nibbles",

"species": "Rabbit",

"yearBought": 2011,

"isAlive": true

}

Keys are always text strings and can be listed in any order. Values can be text (in quotes such as "Nibbles" and "Rabbit" above), numbers (such as 2011 above) or Boolean values (true or false). Importantly, values can also be whole objects. This allows us to build up more complex objects that are themselves composed of objects. For example:

{

"isAlive": true,

"favourites": {

"activity": "digging holes",

"food": "carrots"

},

"species": "Rabbit",

"name": "Nibbles",

"yearBought": 2011

}

Finally, it is possible to group values (text, numbers, Boolean values and objects) together in *arrays*. Arrays are indicated with square brackets, just as we do in Processing and items in the array separated by commas. Unlike the unordered storage of key-value pairs in an object, array elements retain the order in which they are defined in the JSON file. For example we might build up an array of pets where the first array item always refers to the rabbit called Nibbles and the second object a tortoise called Bash:

[

{

"isAlive": true,

"species": "Rabbit",

"name": "Nibbles",

"yearBought": 2011

},

{

"isAlive": true,

"species": "Tortoise",

"name": "Bash",

"yearBought": 1994

}

]

Many JSON files are complex nested collections of objects, arrays and text/numerical values. Thankfully Processing provides a collection of methods for reading, parsing and writing JSON objects and arrays. To see how they can be used, consider the task of mapping the status of London's bikeshare scheme ("Boris Bikes").

The official API for live bike station updates, known as 'BikePoint' is provided by Transport for London at[www.tfl.gov.uk/tfl/syndication/feeds/cycle-hire/livecyclehireupdates.xml](https://tfl.gov.uk/info-for/open-data-users/data-feeds?intcmp=29422). While access is free, you need to sign up with TfL to receive an application ID and key. This allows you to make web requests to the TfL service and receive live data in JSON format. For example to request details about the 700 or so bike docking stations including their name,location current number of bikes docked, you could store the result in a JSON array as follows:

|  |
| --- |
| JSONArray ja =loadJSONArray("<https://api.tfl.gov.uk/BikePoint?app_id=XXX&app_key=YYY>"); |

where XXX and YYY are substituted with the app and key balues provided to you by Tfl after registering. The hard work of parsing the complex JSON file is largely done by processing although you still need to know which key-value pairs you wish to extract.

To keep things simple, the sketch below reads the JSON file not from the TfL feed directly, but from a locally stored copy of the file [bikeStations.json](http://gicentre.org/datavis/session04/bikeStations/data/bikeStations.json).

|  |
| --- |
| // Displays London Bike Hire docking stations by reading a JSON file  // containing station status data.  // Jo Wood, 12th February, 2016.    Table dsTable;  float minLng,maxLng,minLat,maxLat;    void setup()  {    size(900, 500);    textSize(6);    textAlign(CENTER,BOTTOM);    noStroke();      readJSON();        // Reads the docking station details from a JSON file.    minLng = -0.237;   // E-W and N-S range of station locations.    maxLng = -0.002;    minLat = 51.454;    maxLat = 51.550;  }    void draw()  {    background(255);      for (int row=0; row<dsTable.getRowCount(); row++)    {      float x =map(dsTable.getFloat(row,"longitude"),minLng,maxLng,50,width-50);      float y =map(dsTable.getFloat(row,"latitude"),minLat,maxLat,height,10);        fill(80);      text(dsTable.getString(row,"name"),x,y);      fill(120, 50, 50);      ellipse(x, y, 4, 4);    }    noLoop();     // Just draw once.  } |

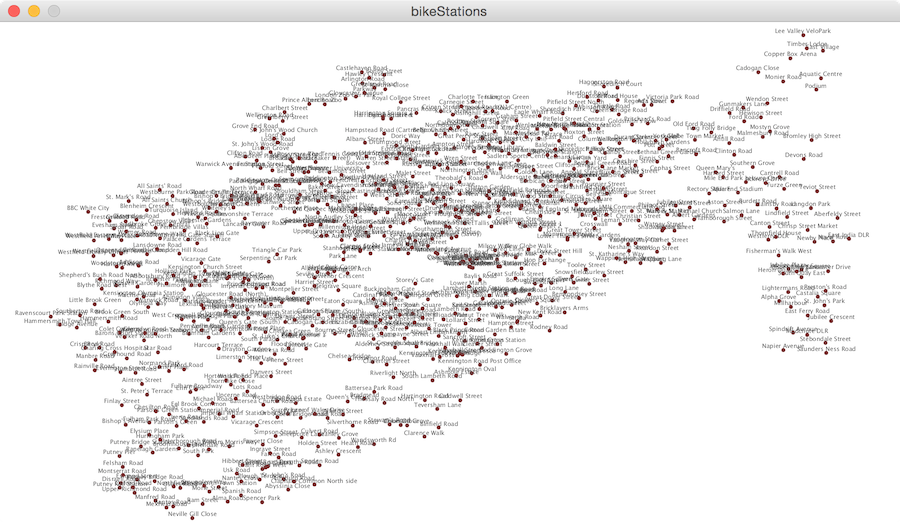
The user-created method readJSON() does the work of extracting the bits of the JSON file we need for our visualization:

|  |
| --- |
| void readJSON()  {    // Create empty table in which to put data    dsTable =new Table();    dsTable.addColumn("name");    dsTable.addColumn("longitude");    dsTable.addColumn("latitude");    dsTable.addColumn("numDocks");    dsTable.addColumn("numBikes");    dsTable.addColumn("numSpaces");      // Load the JSON data which is an array of objects, one for each docking station.    JSONArray jArray =loadJSONArray("bikeStations.json");      for (int i=0; i<jArray.size(); i++)    {      JSONObject dStation = jArray.getJSONObject(i);        // Name, latitude and longitude are all key-value pairs in each JSON object      String fullName = dStation.getString("commonName");      float latitude  = dStation.getFloat("lat");      float longitude = dStation.getFloat("lon");        // Number of docking points, spaces and bikes are stored in an array sub-objects      JSONArray otherProps = dStation.getJSONArray("additionalProperties");      int numBikes  = otherProps.getJSONObject(6).getInt("value");      int numSpaces = otherProps.getJSONObject(7).getInt("value");      int numDocks  = otherProps.getJSONObject(8).getInt("value");        // Add a new row containing data to table.      if (numDocks > 0)      {        TableRow row = dsTable.addRow();        String[] names =split(fullName,",");        row.setString("name", names[0]);        row.setFloat("longitude", longitude);        row.setFloat("latitude", latitude);        row.setInt("numDocks", numDocks);        row.setInt("numBikes", numBikes);        row.setInt("numSpaces", numSpaces);      }    }  } |

This method looks for elements of the JSON file that contain the name, location (longitude and latitude), capacity and number of bikes in each station and extracts them storing them in a Processing Table for use in the main drawing sketch.

The main part of the sketch with the setup() and draw() methods uses the now standard approach to scaling data values (in this case longitude and latitude) with the [map()](http://processing.org/reference/map_.html) command so that each docking station can be drawn within the sketch area as an ellipse with the name of the docking station drawn above it. The only new Processing command is[textSize()](http://processing.org/reference/textSize_.html), which, well, you can guess what it does.

Output from the sketch is a fairly poor representation of the data, but illustrates how we can incorporate JSON parsing into our own visualization sketches and would not be hard to adapt to improve its representation. We will look at improving the visual depiction of the data next week when we consider interaction.

[](http://staff.city.ac.uk/~jwo/datavis/session04/images/bikeStations.png)

Output from bike stations sketch.

## 4.3.2 Parsing XML data

Like JSON, XML files can represent data as *trees* or hierarchies of elements. That is, each *element* in the XML file can have 0 or more *child elements*. In turn, those children can have their own child elements. Let's consider a simple example of parsing the following XML file, [waypoints.gpx](http://moodle.city.ac.uk/mod/page/gpsPlotter/data/waypoints.gpx) (the first few lines only are shown here).

<?xml version="1.0" encoding="UTF-8" standalone="no" ?>

<gpx xmlns="http://www.topografix.com/GPX/1/1" version="1.1"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.topografix.com/GPX/1/1 http://www.topografix.com/GPX/1/1/gpx.xsd">

<rte>

<rtept lat="51.941322" lon="0.22167258">

<name>01strt</name>

<sym>Waypoint</sym>

</rtept>

<rtept lat="51.94083" lon="0.2175779">

<name>02 SO</name>

<sym>Waypoint</sym>

</rtept>

<rtept lat="51.93799" lon="0.19855696">

<name>03 R</name>

<sym>Waypoint</sym>

</rtept>

<rtept lat="51.942554" lon="0.19898307">

<name>04 L</name>

<sym>Waypoint</sym>

</rtept>

</rte>

</gpx>

The XML file represents output from a GPS ('satnav'), using the [GPX format](http://www.topografix.com/gpx.asp). In this case, the *root element* is one called **gpx**which has a child called **rte**. This child in turn has a collection of children, each called **rtept**. Each rtept has further children (name and symbol) as well as *attributes* **lat** and **lon**.

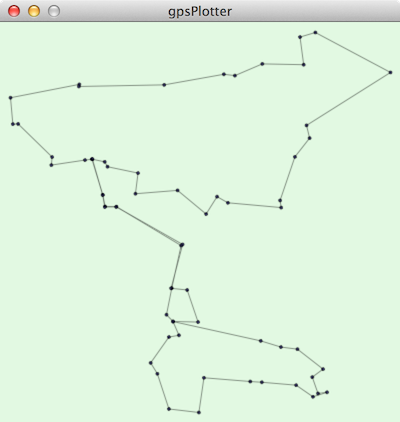
Suppose we wished to write a sketch that read the latitude and longitude values from the GPX file and plotted them. The code to do this is shown below.

|  |
| --- |
| // Reads and displays a set of GPS waypoints.  // Jo Wood, 13th February, 2014    float[] easting,northing;  float minX =MAX\_FLOAT;  float minY =MAX\_FLOAT;  float maxX =MIN\_FLOAT;  float maxY =MIN\_FLOAT;    void setup()  {    size(400,400);    strokeWeight(0.5);      XML xml =loadXML("waypoints.gpx");    XML[] routePoints = xml.getChildren("rte/rtept");    int numPoints = routePoints.length;    easting  =new float[numPoints];    northing =new float[numPoints];      for (int i=0; i<numPoints; i++)    {      easting[i]  = routePoints[i].getFloat("lon");      northing[i] = routePoints[i].getFloat("lat");        minX =min(minX,easting[i]);      maxX =max(maxX,easting[i]);      minY =min(minY,northing[i]);      maxY =max(maxY,northing[i]);    }  }    void draw()  {    background(220,250,220);    beginShape();    for (int i=0; i<easting.length; i++)    {      float x =map(easting[i],minX,maxX,10,width-10);      float y =map(northing[i],minY,maxY,height-10,10);        vertex(x,y);      fill(50,50,100);      ellipse(x,y,3,3);      noFill();    }    endShape();      noLoop();     // Draw only once.  } |

Each XML element is stored in a special variable of type [XML](http://processing.org/reference/XML.html). An XML variable will store its name, any attributes it has and any children it has. In this example, we are interested in the rtept elements, so we have to navigate 'down' the tree by finding the child of the rte element. Once we have found the children, we can loop through each of them and extract the lat and lon attributes as float values.

Each latitude and longitude is then stored in an array of points that are drawn by the draw() method as we have seen before.

Many data sources are provided in XML formats, so it can be useful to familiarise yourself with Processing's [XML](http://www.processing.org/reference/XML.html) class.

[](http://gicentre.org/datavis/session04/images/gps.png)

Output from GPS sketch

## 4.4 Conclusions

Parsing data can be a complicated and time consuming part of data visualization. We have covered a number of approaches that attempt to reduce the effort required to add structure to the data you read. The exact approach taken will depend on the structure of the data (e.g. plain text, tabular data, TSV, CSV, hierarchical markup etc.) as well as what you need to parse in order to produce your visualization.

Tabular text data, JSON files and XML files are the most common for the kinds of data you are likely to wish to visualize, but there are some other more complex data sources out there. The good news is that there are also plenty of Processing libraries that make parsing them easier. A good place to browse these libraries is the **Data** section of the [Processing Libraries Reference](http://processing.org/reference/libraries/) page. This includes, for example libraries for [natural language processing](http://www.rednoise.org/rita/) and [communication with databases](http://bezier.de/processing/libs/sql/) such as SQLite and MySQL.

There has been a lot of material in this session as well as a number of new programming concepts. If you are new to programming, this may feel a little overwhelming at this stage. It does, get easier though! We have now covered most of the programming concepts you need to write even quite sophisticated Processing sketches. From the next session onwards we will be concentrating more on representation and design issues and there will be far less in the way of new programming ideas.

## Recommended Reading

*The second half of Chapters 4 in Fry (2008) covers many of the programming techniques (loops, arrays and methods) discussed in this session along with some simple parsing and representation. For an overview of more advanced parsing approaches including CSV files, XML, databases and spatial data, see Chapter 10 of Fry (2008).*

**Fry, B.** (2008) Chapter 4: Time Series, pp.54-93 in *Visualizing Data*, O'Reilly

**Fry, B.** (2008) Chapter 10: Parsing Data, pp.296-330 in *Visualizing Data*, O'Reilly