CSCE 120: Learning To Code

Introduction To Data

Introduction

This module introduces you to data and various data formats. We discuss what can be done with data including transformation, organization, aggregation, visualization and data mining. We will examine various data *types* used in JavaScript Object Notation (JSON) in preparation for using data in the JavaScript programming language. We will also look at various errors and anomalies that can arise when working with data.

Working With Data

Data is intended to model real-world problems. Consider the data in Table 1 which models enrollment data including some student information and the course(s) in which they are enrolled.

First Name	Last Name	NUID	Email	Year	GPA	Course Number	Course Name
Starlin	Castro	12301013		Sophomore	3.75	CSCE 120	Learning To Code
Starlin	Castro	12301013		Sophomore	3.75	MATH 103	College Algebra & Trigonometry
Starlin	Castro	12301013		Sophomore	3.75	MUNM 287	History of Rock Music
Anthony	Rizzo	44001244	arizzo@mlb.com	Freshman	3.24	CSCE 120	Learning To Code
Anthony	Rizzo	44001244	arizzo@mlb.com	Freshman	3.24	MUNM 287	History of Rock Music
Anthony	Rizzo	44001244	arizzo@mlb.com	Freshman	3.24	PSYC 181	Introduction to Psychology
Edwin	Jackson	00321023	ejackson@unl.edu	Senior	2.95	MRKT 257	Sales Communication
Edwin	Jackson	00321023	ejackson@unl.edu	Senior	2.95	FINA 260	Personal Finance
Brett	Jackson	93213394	brett.jackson@gmail.com	Freshman	3.8	CSCE 120	Learning To Code
Brett	Jackson	93213394	brett.jackson@gmail.com	Freshman	3.8	BLAW 372	Business Law I
Javier	Baez	33928192	jbaez@cubs.com	Freshman	3.21	ECON 211	Principles of Macroeconomics
Javier	Baez	33928192	jbaez@iacubs.com	Freshman	3.21	BLAW 372	Business Law I
Javier	Baez	33928192	jbaez@iacubs.com	Freshman	3.21	ENGL 150	Writing: Rhetoric as Inquiry
Junior	Lake	11223344	j_lake@yahoo.com	Sophomore	2.81	MUNM 287	History of Rock Music
Junior	Lake	11223344	jlake@gmail.com	Sophomore	2.81	CSCE 120	Learning To Code
Richard	Renteria	89320191	drenteria@gmail.com	Senior	3.91	ENGR 100	Interpersonal Skills for Engineering Leaders
Richard	Renteria	89320191	drenteria@gmail.com	Senior	3.91	CSCE 120	Learning To Code
Richard	Renteria	89320191	rrenteria@cubs.com	Senior	3.91	PHYS 211	General Physics I
Ryne	Sandberg	33221232	sandberg@mlb.com	Junior	3.45	BLAW 300	Business, Government & Society
Ryne	Sandberg	33221232	sandberg@mlb.com	Junior	3.45	CSCE 477	Cryptography & Security

Table 1: Student Enrollment Data

```
firstName,lastName,nuid,email,year,gpa,courseNumber,courseName
Starlin,Castro,12301013,scastro@cubs.com,Sophomore,3.75,CSCE 120,Learning To Code
Starlin,Castro,12301013,scastro@cubs.com,Sophomore,3.75,MATH 103,College Algebra & Trigonometry
Starlin,Castro,12301013,scastro@cubs.com,Sophomore,3.75,MUNM 287,History of Rock Music
Anthony,Rizzo,44001244,arizzo@mlb.com,Freshman,3.24,CSCE 120,Learning To Code
Anthony,Rizzo,44001244,arizzo@mlb.com,Freshman,3.24,MUNM 287,History of Rock Music
Anthony,Rizzo,44001244,arizzo@mlb.com,Freshman,3.24,PSYC 181,Introduction to Psychology
Edwin,Jackson,00321023,ejackson@unl.edu,Senior,2.95,MRKT 257,Sales Communication
Edwin,Jackson,00321023,ejackson@unl.edu,Senior,2.95,FINA 260,Personal Finance
Brett,Jackson,93213394,brett.jackson@gmail.com,Freshman,3.8,CSCE 120,Learning To Code
Brett,Jackson,93213394,brett.jackson@gmail.com,Freshman,3.8,BLAW 372,Business Law I
Javier,Baez,33928192,jbaez@cubs.com,Freshman,3.21,ECON 211,Principles of Macroeconomics
Javier,Baez,33928192,jbaez@iacubs.com,Freshman,3.21,BLAW 372,Business Law I
Javier,Baez,33928192,jbaez@iacubs.com,Freshman,3.21,ENGL 150,Writing: Rhetoric as Inquiry
Junior,Lake,11223344,j_lake@yahoo.com,Sophomore,2.81,MUNM 287,History of Rock Music
Junior,Lake,11223344,jlake@gmail.com,Sophomore,2.81,CSCE 120,Learning To Code
Richard,Renteria,89320191,drenteria@gmail.com,Senior,3.91,ENGR 100,Interpersonal Skills for Engineering Leaders
Richard,Renteria,89320191,drenteria@gmail.com,Senior,3.91,CSCE 120,Learning To Code
Richard,Renteria,89320191,rrenteria@cubs.com,Senior,3.91,PHYS 211,General Physics I
Ryne,Sandberg,33221232,sandberg@mlb.com,Junior,3.45,BLAW 300,"Business, Government & Society"
Ryne,Sandberg,33221232,sandberg@mlb.com,Junior,3.45,CSCE 477,Cryptography & Security
```

Figure 1: A CSV data format

What can we do with this data? That is, what kind of operations can we perform on the data? The data itself is just raw information. Raw information is like unrefined ore: we need to process it in order to get separate the useless (or at least less-useful) material from the valuable material. What kind of new insights or new information can we gain by processing this data?

Data Transformation & Formatting

A simple operation that can be performed is *data transformation* which involves simply translating the data from one *format* into another format so that various programs and programming languages can more easily process and "recognize" the data. For example, we could view the data in Table 1 as a "flat" data view: each column represents one piece of data or a *field* while each row represents a single record.

A table is good for a human-readable representation of some data. However, how is data actually stored in a computer? One format is to use a Comma Separated Value (CSV) data format which is simply a plain text file with one record (row) per line. Each piece of data is separated by a single comma. An example can be viewed in Figure 1. This format is less human-readable, but it is easy for a computer program to parse and "read."

Alternatively, this data could be stored in a spreadsheet program such as Microsoft's Excel. Such programs have their own, proprietary format and their own way to internally represent the data.

Other, more "open" data format standards include XML (Extensible Markup Language) which represents data by marking each piece of data with a *tag* that semantically defines what that piece of data represents. In XML, a tag is denoted with angle brackets, `<tagName>`. Each opening tag *must* be closed by a corresponding closing tag, `</tagName>`. An abridged example can be found in Figure 2. Note the *structure* of the data: each tag has a *nested* collection of other tags. This defines a data "tree" in which nested elements are the "children" of the "parent" element. This structure allows us to

identify and infer relationships between data.

```xml
<?xml version="1.0"?>
<roster>
    <enrollment>
        <firstName>Starlin</firstName>
        <lastName>Castro</lastName>
        <nuid>12301013</nuid>
        <email>scastro@cubs.com</email>
        <year>Sophomore</year>
        <gpa>3.75</gpa>
        <courseNumber>CSCE 120</courseNumber>
        <courseName>Learning To Code</courseName>
    </enrollment>
    <enrollment>
        <firstName>Starlin</firstName>
        <lastName>Castro</lastName>
        <nuid>12301013</nuid>
        <email>scastro@cubs.com</email>
        <year>Sophomore</year>
        <gpa>3.75</gpa>
        <courseNumber>MATH 103</courseNumber>
        <courseName>College Algebra &amp;amp; Trigonometry</courseName>
    </enrollment>
    ...
    <enrollment>
        <firstName>Ryne</firstName>
        <lastName>Sandberg</lastName>
        <nuid>33221232</nuid>
        <email>sandberg@mlb.com</email>
        <year>Junior</year>
        <gpa>3.45</gpa>
        <courseNumber>CSCE 477</courseNumber>
        <courseName>Cryptography &amp;amp; Security</courseName>
    </enrollment>
</roster>
```

Figure 2: XML Formatted Data

Another open format and the one we'll work with is JavaScript Object Notation (JSON). Similar to XML, JSON has a nested structure. However, it is less verbose as it does not require opening/closing tags to identify data. As such, JSON is generally considered a more "light-weight" data format as the same data can be represented with fewer characters and thus a smaller file size resulting in less storage and less transmission time when sent over a network. An example can be found in Figure 3; we examine JSON formatting in detail in later sections.

Translating from one format to another is a common Electronic Data Interchange (EDI) problem. Often, different systems written in different languages and different technologies need to communicate with each other. By exchanging data in a common, standardized format, communication can easily take place.

Data Organization

Transforming and translating data facilitates communication, but it does not necessarily give us greater insight as to what the data represents. Instead, we need to think of

```
{
  "roster": [
  {
    "firstName":"Starlin",
    "lastName":"Castro",
    "nuid":12301013,
    "email":"scastro@cubs.com",
    "year":"Sophomore",
    "gpa":3.75,
    "courseNumber":"CSCE 120",
    "courseName":"Learning To Code"
  },
  {
    "firstName":"Starlin",
    "lastName":"Castro",
    "nuid":12301013,
    "email":"scastro@cubs.com",
    "year":"Sophomore",
    "gpa":3.75,
    "courseNumber":"MATH 103",
    "courseName":"College Algebra & Trigonometry"
  },
  ...
  {
    "firstName":"Ryne",
    "lastName":"Sandberg",
    "nuid":33221232,
    "email":"sandberg@mlb.com",
    "year":"Junior",
    "gpa":3.45,
    "courseNumber":"CSCE 477",
    "courseName":"Cryptography & Security"
  }
  ]
}
```

Figure 3: JSON Formatted Data

more sophisticated operations. For example, we may want to *sort* the data and/or *search* the data for particular records. For example, we may want to sort and *filter* records to produce a course schedule for a particular student or to produce a roster for a particular class. This necessarily means that we have to process the data in a particular manner.

Even a simple operation of (re)sorting the data involves a lot of details. For example: how do you want to sort? By last name-first name? By GPA? For entries with the same person, how should the courses be sorted? Alphabetically? According to when the course was taken? By grade received? Depending on your *use case* (how the user will actually use the processed data and for what) you could have very different answers to each of these questions.

There are also issues involving efficiency. There are many sorting and searching algorithms to choose from. Do they scale? Our data is rather small with a few dozen records but in a real system you may have thousands, millions, billions of records. How can such data be organized so that a simple operation of a student searching for their classes can be done in milliseconds rather than even seconds.

Data organization also involves *data normalization*. The data in Table 1 is "flat" in that there are lots of repeated entries. A student may be enrolled in 5 courses, but is it really

necessary to repeat their name, NUID, etc. for *each* of those entries? Moreover, this could lead to *errors* and data anomalies: what if a user updates their email? How many records must be effected? What if one of them is accidentally omitted?

Normalization is the process of *separating* data into different tables and *relating* records between them so that repetition is minimized and data anomalies are prevented or at least mitigated. This is what is done when working with a Relational Database Management system (RDBM) such as MySQL or PostgreSQL.

Data Aggregation

Data aggregation is a process by which we can group and combine data to produce statistics and other information. A simple example could include counting the number of (unique) students or courses in our enrollment records. Or we could compute the average number of credit hours per student. We could even consider individual students: we may be interested in the total number of credit hours for *each* student in order to identify those who are under or over-enrolled. Such operations may require us to *group* pieces of data together (by-student or by-course) and *project* or "flatten" the data out to take a sum, average, count, etc.

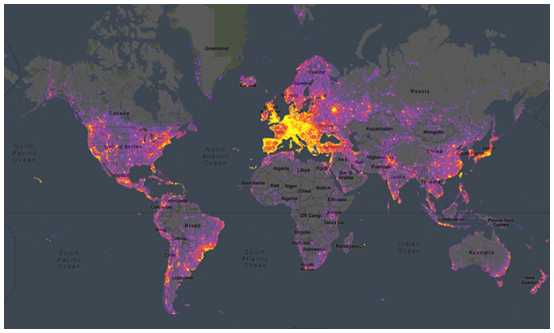
Data Visualization

Data is useless as long as it is simply sitting in a computer. In order to be useful, it needs to inform a decision or give us (humans) better insight into a problem. The data formats above are great for a computer but terrible for human consumption. It is difficult to make any sense or overall impression of such raw data.

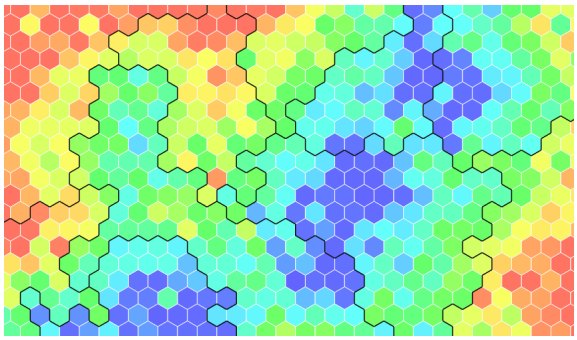
Instead, for human consumption, data is usually *visualized* in some manner so that it can easily be interpreted by a human user. This can be as simple as a bar graph or more complex such as a heat map or connection graph as in Figures 4 and 5. These data visualizations immediately allow us to discern patterns and discover insights that the raw data would not have otherwise revealed. In fact, humans are much better, in general, at recognizing these patterns than computers are.

Data Mining

More advanced data processing techniques includes cutting edge *data mining* and *machine learning* which include very sophisticated algorithms and statistical techniques that can process data and "learn" new patterns. Basic data processing may involve knowing what questions you want answered (how many students, how many credit hours, etc.). These advanced techniques, however, can be used to answer questions that you had no idea that you were even interested in. Advanced analysis can be used to find patterns and trends and extract new information and insights.

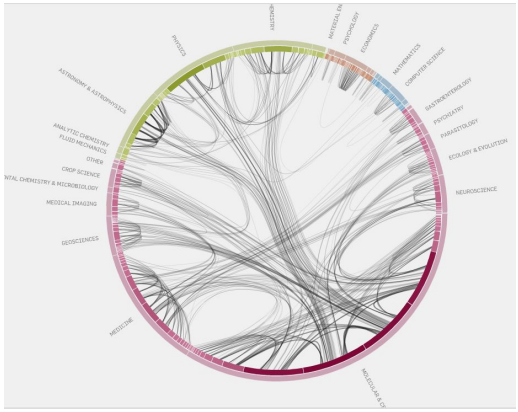


(a) Geographic Heatmap

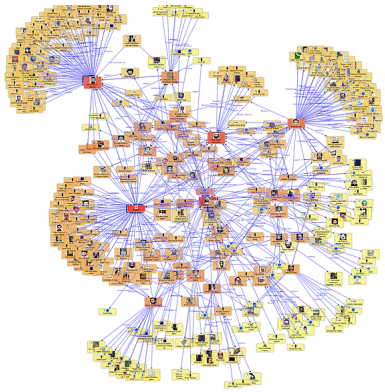


(b) Data Heatmap

Figure 4: Heat Map Data Visualizations



(a) Connection Graph



(b) Social Connection Graph

Figure 5: Connection Graph Data Visualizations

JavaScript Object Notation (JSON)

The data format that we'll focus on is JavaScript Object Notation or simply just JSON. The format itself is a subset of the JavaScript programming language and as such it has the advantaged that it is recognized and "built-in" to the JavaScript language. In JSON, basic data are represented as *objects* (or "entities" or "things") which contain key-value pairs of data. The key is a unique *string* that is used to access the data while the value is the content that is stored in the data. A *string* is simply a collection of alphanumeric characters. Key value pairs are denoted using a string-colon-value syntax: `"key": value` where `value` can be one of several data *types* which we explore in detail below.

Objects are denoted with opening and closing curly brackets, `{...}`. Within the object, key-value pairs are enumerated as a comma-delimited list. A full example can be seen in Figure 6.

Within each object, the key is considered *unique*. Duplicate keys are not an error, however

```json
{
  "student": {
    "firstName": "John",
    "lastName": "Student",
    "nuid": 12345678,
    "gpa": 3.85,
    "emails": ["jstudent@unl.edu", "johnny@gmail.com"]
  },
  "course": {
    "id": 4231,
    "name": "Learning to Code",
    "code": "CSCE 120"
  }
}
```

Figure 6: A Full JSON Example

whichever key-value pair appears *last* is the one that is used. All prior duplicates are effectively ignored. For example in the following object,

```json
{
  "foo": "bar",
  "foo": "baz",
}
```

the key `foo` is defined twice. The second value, `"baz"` is the value that is ultimately used while the value `"bar"` is effectively ignored.

Note:

- Only strings may be used for keys, you cannot use numbers or other types as keys.
- Keys (like all strings) are *case-sensitive* so `"foo": 1` is not the same thing as `"Foo": 1`
- Though you can use spaces and other special characters in your key string, it is generally bad practice to do so. Instead, use a modern convention such as `"lowerCamelCasing"` in which keys with multiple words are written together with the first word of each subsequent word capitalized and all other letters lowercased.

JSON Data Types

There are four basic types of data that are supported by JSON.

Numbers

Numeric types can either be integers (whole numbers) like $123, 0, -132$ or floating point numbers $3.14, 8.90, -0.0321$. In JSON, the numeric types are denoted as numeric *literals*. Scientific notation is supported using either `e` or `E`. Some examples:

```
{
  "pi": 3.14,
  "two": 2,
  "billion": 1e9,
  "nano": 1e-9,
  "numbers": [1, 2, 3, 4.5, -10, 0]
}
```

Strings

A *string* is a collection of ordered characters. Characters can include alphabetic characters (upper and lower case), numeric characters (which are not treated as numbers), as well as whitespace, punctuation, etc. Strings can consist of printable characters as well as non-printable control characters. Strings can also contain international characters (UTF-8) to represent characters in other languages that do not use the Latin alphabet.

In JSON strings are denoted by double quotes, `"Hello World!"`. Everything inside the beginning/ending double quotes is considered part of the string including space(s) and punctuation Strings can hold any printable (and some unprintable) characters as well as international (UTF-8) characters.

Some special characters need to be *escaped* by placing a backslash, `\`, in front of them. The most common special characters are `\", \\, \n, \t` (double quotes, backslash, end line, and tab). In addition, UTF-8 characters are denoted with a `\u` followed by 4 hexadecimal characters corresponding to the special character. Some examples:

```
{
  "hello": "world",
  "nickname": "John \"The Man\" Student",
  "version": "2.1.2",
  "description": "Introduction to stembolt repair \nsecond edition",
  "Japan": "\u65E5\u672C",
}
```

The unicode characters in the example could be rendered as 日本, which represents the Japanese writing for the country of Japan.

Booleans

A *boolean* value is a value that is either true or false. In JSON, the keywords `true` and `false` are special keywords that are used to denote these values. They are *not* surrounded by double quotes as they are not strings.

Booleans are used to model *flags* or properties that can either be true or false rather than hold a value such as a string or number. Some examples:

```
{
  "isGrad": false,
  "isHonors": true,
}
```

Arrays

Arrays are a way to collect pieces of data into one *ordered* collection. In JSON, arrays are denoted with opening and closing square brackets. Elements in an array are delimited using commas. Moreover, the elements are *ordered*: the first element in the list is the first element in the array, the second in the list is the second element, etc. Ultimately the order may or may not matter, but the order in which you list elements in an array is the order that they will be stored in the array. Informally, an array can be viewed as the collection of *rows* in a table: a collection of records.

Some examples:

```
{
  "names": ["John C. Reilly", "Jacob O'Brien", "Steve McQueen"],
  "firstNames": ["joe", "jane", "Joe", "Jane"],
  "ids": [321, 2321, 9392, 89122],
  "scores": [90.1, 83.2, 78.24]
}
```

In general, different types can be stored in the same array. For example, one array may hold a number as its first element, a string as its second element, and an object as its third element. It may even hold another array! However, this kind of *type mixing* is discouraged in practice. If different types need to be collected together, then it may be more appropriate to collect them in an object.

Objects

Unlike an array, an *object* is an unordered collection (a set) of key-value data. In contrast to arrays, objects can be viewed as the collection of columns for one particular row (record). An object essentially *encapsulates* (groups) several pieces of data together into one logical entity.

An object is denoted using opening and closing curly bracket. Anything placed within these brackets is part of the object. Objects can be *nested* so that objects can be comprised of key-value data pairs whose values are *also* objects. This is known as *composition*: one object is composed of a collection of other objects.

One special object is the *root* object. As in all previous examples, when we begin a JSON object, the opening and closing curly brackets represent a value (an object), but they do not have a key. It is understood that this represents the root object. The name root derives from the idea that like XML, a JSON object represents a tree with all child/descendant elements emanating from the root element.

The JSON example in Figure 6 is visualized as a tree in Figure 7 and as a nested table in 8.

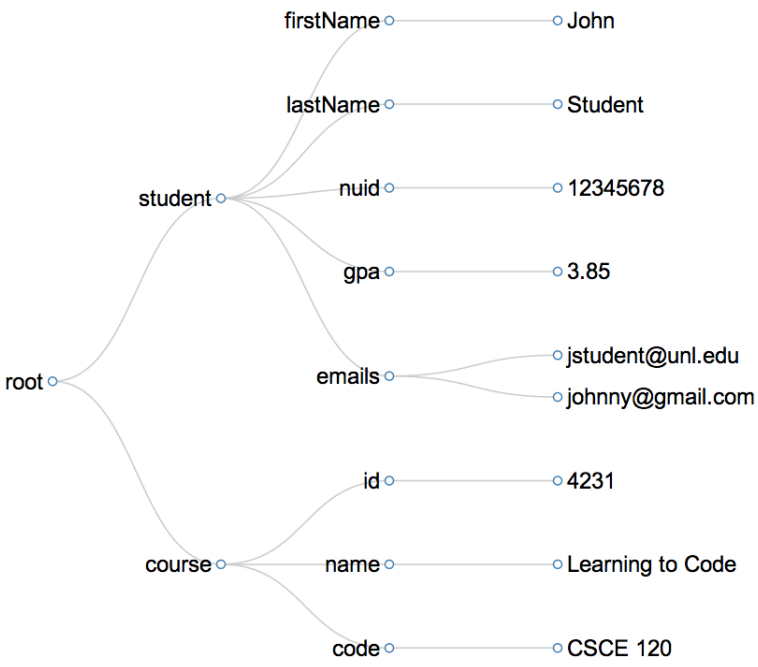


Figure 7: A tree visualization of the JSON object

The Null Value

Another special keyword, `null` is used to denote missing, unknown, undefined or non-applicable data. The `null` keyword can be used for any key-value pair an the type is necessarily undefined–its not a string, array, number, etc.; it is `null`. An example:

```
{
  "gpa": null,
  "gradCourses": null
}
```

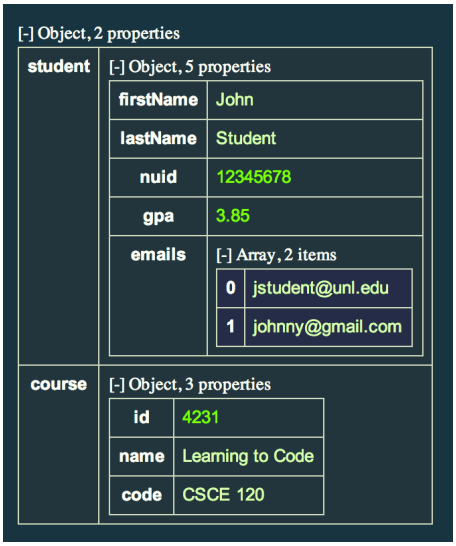


Figure 8: A visualization of the JSON object, image generated by http://chris.photobooks.com/json/.

There is a distinct difference between not having a key-value pair and having a key-value pair with a value of `null`. In the first case the key-value pair itself does not exist. In the second case, the pair exists, but the value is `null`. That is, the *key* exists, but the value does not.

JSON Formatting

In addition to the syntax rules already presented, there are some other rules that JSON data has to follow. All brackets must be *well-balanced*. That is, for every opening curly bracket there is a corresponding closing curly bracket. Likewise for square brackets and double quotes. Failure to close a section with the proper closing bracket will render your JSON invalid. Moreover, delimiter types cannot be mixed and must be *nested* correctly. You cannot interleave different types such as `{ ... [ ... } ... ]`. In this example, each opening bracket has a match, but they are not nested properly.

All whitespace outside of a double quote in JSON is essentially meaningless data. In the examples above, we separated each element on its own line and indented properly according to the nesting of elements. This is sometimes referred to as "pretty printing": formatting the JSON data so that it is easily human readable.

This is nice for development, but when it comes to processing the data, machines ultimately ignore the extra whitespace. You can imagine that if we always transmitted this extra whitespace, we would be wasting a lot of bandwidth, storage, and processing power. For that reason, data is usually stored/transmitted without the extra whitespace. Sometimes this is referred to as a *compact* representation. The example from Figure 6 could be more compactly written as follows (note that the data intentionally runs off the page as line breaks are unnecessary).

{"student":{"firstName":"John","lastName":"Student","nuid":12345678,"gpa":3.85,"email

Data Errors

JSON is a data interchange format intended to be processed by computers and programs. Though as developers we may read and work with JSON data, the end user will never interact with JSON directly. Humans are relatively tolerant of many formatting errors: though a text may contain spelling and punctuation errors, we can still, within reason, read it and grasp the general intention of the writer.

In contrast, machines are dumb: they don't possess the same reasoning that we do and are not, in general, able to process data containing various errors. Some errors are completely fatal and result in programs being unable to execute. Programs may be able to tolerate other errors to a point, meaning that they may execute but with unexpected or unintended results. Yet other errors may not result in any unexpected results but are still considered (say by the end user) to be nonsense resulting from "bad" (though not erroneous) data. Such a situation is often referred to as garbage-in garbage-out: the program may run as intended, but since we gave it garbage data, it gave us garbage results.

Formatting Errors

As previously mentioned, valid JSON must be in a certain format. Special characters must be escaped, brackets must be well-balanced, delimiters (commas, colons, etc.) must be used properly. Invalid JSON cannot even be processed by a program as it does not conform to the specified rules.

Syntax Errors

Other errors may occur in JSON that are not formatting errors. The JSON data could be well-formatted and valid, but other syntax errors may cause the program that processes the data to fail. A prime example could be a misspelling or unexpected formatting of keys. Recall that keys may contain spaces and still be valid, but it is usual to use naming conventions such as lower camel casing where multi-word keys are written with no spaces and each new word (except for the first) is capitalized.

When code is written to process JSON data it is written with certain expectations that, when violated, cause errors. A misspelling or incorrect use of spaces, or case sensitivity issue may be the culprit.

Consistency Errors

JSON data could be well-formatted and free of syntax errors, but it may still contain "bad" data. That is, the data is readable and can be processed without error, but the result that the end user sees is invalid.

Examples of such errors can include:

- Numeric values that are outside the range of valid values
- Misspellings of data values
- Inconsistent spelling or formatting of data values
- Misidentified data items (such as a switched first or last name)
- Inconsistent data for the "same" logical entity (a person's name appears as Joe in one record, but Joseph in another though they are the same person)

Normalization

The last type of data error identified above can be solved with data normalization by separating out pieces of data into their own logical entities and defining *relations* between these records. For example, a roster may consist of course data being repeated for each person enrolled in that course. This opens the potential for the kind of consistency errors identified above. However, a normalized version of this data would have a single course record for each actual course with a unique identifier. Enrollment would then be modeled by, say, having each student record own a list of course IDs for courses in which they are enrolled.