

# How to Parse JSON Files on the Linux Command Line with jq



**DAVE MCKAY** [@thegurkha](#)  
FEBRUARY 14, 2020, 8:00AM EDT



Fatmawati Achmad Zaenuri/Shutterstock

JSON is one of the most popular formats for transferring text-based data around the web. It's everywhere, and you're bound to come across it. We'll show you how to handle it from the Linux command line using the `jq` command.

## JSON and jq

JSON stands for [JavaScript Object Notation](#). It's a scheme that allows data to be encoded into plain text files, in a self-describing way. There are no comments in a JSON file—the contents should be self-explanatory. Each data value has a text string called a “name” or “key.” This tells you what the data value is. Together, they're known as name:value pairs, or key:value pairs. A colon (:) separates a key from its value.

An “object” is a collection of key:value pairs. In a JSON file, an object begins with an open curly brace ( { ) and ends with a closing brace ( } ). JSON also supports “arrays,” which are ordered lists of values. An array begins with an opening bracket ( [ ) and ends with a closing one ( ] ).

From these simple definitions, of course, arbitrary complexity can arise. For example, objects can be nested within objects. Objects can contain arrays, and arrays can also contain objects. All of which can have open-ended levels of nesting.

In practice, though, if the layout of JSON data is convoluted, the design of the data layout should probably use a rethink. Of course, if you're not generating the JSON data, just trying to use it, you have no say in its layout. In those cases, unfortunately, you just have to deal with it.

Most programming languages have libraries or modules that allow them to parse JSON data. Sadly, the Bash shell [has no such functionality](#).

Necessity being the mother of invention, though, the jq utility was born! With jq, we can [easily parse JSON](#) in the Bash shell. And it doesn't matter whether you have to work with well-engineered, elegant JSON, or the stuff nightmares are made of.

## How to Install jq

We had to install jq on all the Linux distributions we used to research this article.

To install jq on Ubuntu type this command:

```
sudo apt-get install jq
```

```
dave@howtogeek:~$ sudo apt-get install jq
```

To install jq on Fedora, type this command:

```
sudo dnf install jq
```

```
[dave@localhost ~]$ sudo dnf install jq
```

To install jq on Manjaro, type this command:

```
sudo pacman -Sy jq
```

```
[dave@howtogeek ~]$ sudo pacman -Sy jq
```

# How to Make JSON Readable

JSON doesn't care about white space, and layout doesn't affect it. As long as it [follows the rules of JSON grammar](#), systems that process JSON can read and understand it. Because of this, JSON is often transmitted as a simple, long string, without any consideration of layout. This saves a bit of space because tabs, spaces, and new-line characters don't have to be included in the JSON. Of course, the downside to all this is when a human tries to read it.

Let's pull a short JSON object from the [NASA](#) site that [tells us the position](#) of the [International Space Station](#). We'll use `curl`, which can [download files](#) to retrieve the JSON object for us.

We don't care about any of the status messages `curl` usually generates, so we'll type the following, using the `-s` (silent) option:

```
curl -s http://api.open-notify.org/iss-now.json
```

```
dave@howtogeek:~$ curl -s http://api.open-notify.org/iss-now.json
{"iss_position": {"latitude": "-30.8940", "longitude": "-109.8211"}, "timestamp": 1579867008, "message": "success"}dave@howtogeek:~$
```

Now, with a bit of effort, you can read this. You have to pick out the data values, but it isn't easy or convenient. Let's repeat this, but this time we'll pipe it through `jq`.

`jq` uses filters to parse JSON, and the simplest of these filters is a period (`.`), which means "print the entire object." By default, `jq` [pretty-prints](#) the output.

We put it all together and type the following:

```
curl -s http://api.open-notify.org/iss-now.json | jq .
```

```
dave@howtogeek:~$ curl -s http://api.open-notify.org/iss-now.json | jq
{
  "iss_position": {
    "latitude": "-43.7778",
    "longitude": "-90.1201"
  },
  "message": "success",
  "timestamp": 1579867334
}
dave@howtogeek:~$
```

That's much better! Now, we can see exactly what's going on.

The entire object is wrapped in curly braces. It contains two key:name pairs: message and timestamp. It also contains an object called iss\_position, which contains two key:value pairs: longitude and latitude.

We'll try this once more. This time we'll type the following, and redirect the output into a file called "iss.json":

```
curl -s http://api.open-notify.org/iss-now.json | jq . >
```

```
cat iss.json
```

```
dave@howtogeek:~$ curl -s http://api.open-notify.org/iss-now.json | jq
. > iss.json
dave@howtogeek:~$ cat iss.json
{
  "iss_position": {
    "latitude": "-45.0096",
    "longitude": "-87.2543"
  },
  "message": "success",
  "timestamp": 1579867373
}
dave@howtogeek:~$
```

This gives us a well laid out copy of the JSON object on our hard drive.

**RELATED:** [How to Use curl to Download Files From the Linux Command Line](#)

## Accessing Data Values

As we saw above, jq can extract data values being piped through from JSON. It can also work with JSON stored in a file. We're going to work with local files so the command line isn't cluttered with curl commands. This should make it a bit easier to follow.

The simplest way to extract data from a JSON file is to provide a key name to obtain its data value. Type a period and the key name without a space between them. This creates a filter from the key name. We also need to tell jq which JSON file to use.

We type the following to retrieve the message value:

```
jq .message iss.json
```

```
dave@howtogeek:~$ jq .message iss.json
"success"
dave@howtogeek:~$
```

jq prints the text of the message value in the terminal window.

If you have a key name that includes spaces or punctuation, you have to wrap its filter in quotation marks. Care is usually taken to use characters, numbers, and underscores only so the JSON key names are not problematic.

First, we type the following to retrieve the timestamp value:

```
jq .timestamp iss.json
```

```
dave@howtogeek:~$ jq .timestamp iss.json
1579867373
dave@howtogeek:~$
```

The timestamp value is retrieved and printed in the terminal window.

But how can we access the values inside the `iss_position` object? We can use the JSON dot notation. We'll include the `iss_position` object name in the "path" to the key value. To do this, the name of the object the key is inside will precede the name of the key itself.

We type the following, including the `latitude` key name (note there are no spaces between "`iss_position`" and "`latitude`"):

```
jq .iss_position.latitude iss.json
```

```
dave@howtogeek:~$ jq .iss_position.latitude iss.json
"-45.0096"
dave@howtogeek:~$
```

To extract multiple values, you have to do the following:

- List the key names on the command line.

- Separate them with commas (,).
- Enclose them in quotation marks (") or apostrophes (').

With that in mind, we type the following:

```
jq ".iss_position.latitude, .timestamp" iss.json
```

```
dave@howtogeek:~$ jq ".iss_position.latitude, .timestamp" iss.json
"-45.0096"
1579867373
dave@howtogeek:~$
```

The two values print to the terminal window.

## Working with Arrays

Let's grab a different JSON object from NASA.

This time, we'll use a [list of the astronauts who are in space right now](#):

```
curl -s http://api.open-notify.org/astros.json
```

```
dave@howtogeek:~$ curl -s http://api.open-notify.org/astros.json
{"people": [{"name": "Christina Koch", "craft": "ISS"}, {"name": "Alex
ander Skvortsov", "craft": "ISS"}, {"name": "Luca Parmitano", "craft":
"ISS"}, {"name": "Andrew Morgan", "craft": "ISS"}, {"name": "Oleg Skr
ipochka", "craft": "ISS"}, {"name": "Jessica Meir", "craft": "ISS"}],
"number": 6, "message": "success"}dave@howtogeek:~$
```

Okay, that worked, so let's do it again.

We'll type the following to pipe it through jq and redirect it to a file called "astro.json":

```
curl -s http://api.open-notify.org/astros.json | jq . > a
```

```
dave@howtogeek:~$ curl -s http://api.open-notify.org/astros.json | jq
. > astro.json
```

Now let's type the following to check our file:

```
less astro.json
```

```
dave@howtogeek:~$ less astro.json
```

As shown below, we now see the list of astronauts in space, as well as their spacecrafts.

```
{
  "people": [
    {
      "name": "Christina Koch",
      "craft": "ISS"
    },
    {
      "name": "Alexander Skvortsov",
      "craft": "ISS"
    },
    {
      "name": "Luca Parmitano",
      "craft": "ISS"
    },
    {
      "name": "Andrew Morgan",
      "craft": "ISS"
    },
    {
      "name": "Oleg Skripochka",

```

This JSON object contains an array called `people`. We know it's an array because of the opening bracket (`[`) (highlighted in the screenshot above). It's an array of objects that each contain two key:value pairs: `name` and `craft`.

Like we did earlier, we can use the JSON dot notation to access the values. We must also include the brackets (`[]`) in the name of the array.

With all that in mind, we type the following:

```
jq ".people[].name" astro.json
```

```
dave@howtogeek:~$ jq ".people[].name" astro.json
"Christina Koch"
"Alexander Skvortsov"
"Luca Parmitano"
"Andrew Morgan"
"Oleg Skripochka"
"Jessica Meir"
dave@howtogeek:~$
```

This time, all the name values print to the terminal window. What we asked jq to do was print the name value for every object in the array. Pretty neat, huh?

We can retrieve the name of a single object if we put its position in the array in the brackets ([]) on the command line. The array uses [zero-offset indexing](#), meaning the object in the first position of the array is zero.

To access the last object in the array you can use -1; to get the second to last object in the array, you can use -2, and so on.

Sometimes, the JSON object provides the number of elements in the array, which is the case with this one. Along with the array, it contains a key:name pair called number with a value of six.

The following number of objects are in this array:

```
jq ".people[1].name" astro.json
```

```
jq ".people[3].name" astro.json
```

```
jq ".people[-1].name" astro.json
```

```
jq ".people[-2].name" astro.json
```

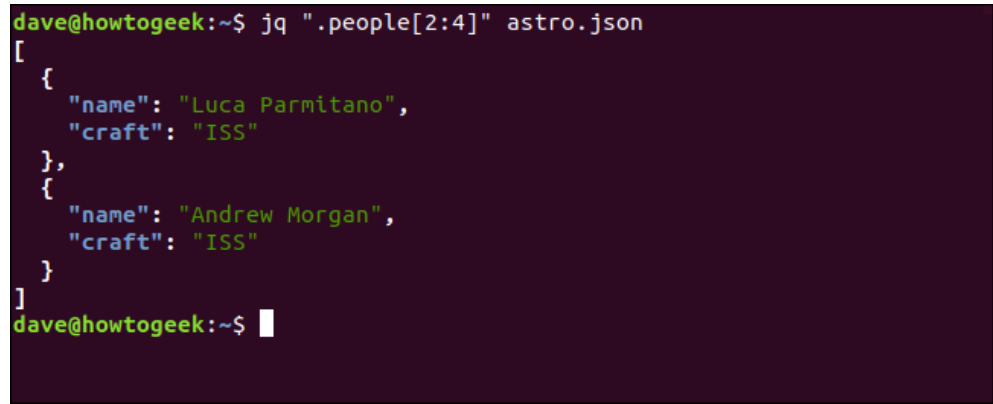
```
dave@howtogeek:~$ jq ".people[1].name" astro.json
"Alexander Skvortsov"
dave@howtogeek:~$ jq ".people[3].name" astro.json
"Andrew Morgan"
dave@howtogeek:~$ jq ".people[-1].name" astro.json
"Jessica Meir"
dave@howtogeek:~$ jq ".people[-2].name" astro.json
"Oleg Skripochka"
dave@howtogeek:~$
```



You can also provide a start and end object within the array. This is called “slicing,” and it can be a little confusing. Remember the array uses a zero-offset.

To retrieve the objects from index position two, up to (but not including) the object at index position four, we type the following command:

```
jq ".people[2:4]" astro.json
```



```
dave@howtogeek:~$ jq ".people[2:4]" astro.json
[
  {
    "name": "Luca Parmitano",
    "craft": "ISS"
  },
  {
    "name": "Andrew Morgan",
    "craft": "ISS"
  }
]
dave@howtogeek:~$
```

This prints the objects at array index two (the third object in the array) and three (the fourth object in the array). It stops processing at array index four, which is the fifth object in the array.

The way to better understand this is to experiment on the command line. You’ll soon see how it works.

## How to Use Pipes with Filters

You can [pipe the output](#) from one filter to another, and you don’t have to learn a new symbol. The same as the Linux command line, jq uses the vertical bar (|) to represent a pipe.

We’ll tell jq to pipe the people array into the .name filter, which should list the names of the astronauts in the terminal window.

We type the following:

```
jq ".people[] | .name" astro.json
```

```
dave@howtogeek:~$ jq ".people[] | .name" astro.json
"Christina Koch"
"Alexander Skvortsov"
"Luca Parmitano"
"Andrew Morgan"
"Oleg Skripochka"
"Jessica Meir"
dave@howtogeek:~$
```

RELATED: [How to Use Pipes on Linux](#)

## Creating Arrays and Modifying Results

We can use jq to create new objects, such as arrays. In this example, we'll extract three values and create a new array that contains those values. Note the opening ([]) and closing brackets ([]) are also the first and last characters in the filter string.

We type the following:

```
jq "[.iss-position.latitude, iss_position.longitude, .tim
```

```
dave@howtogeek:~$ jq "[.iss_position.latitude, .iss_position.longitude
, .timestamp]" iss.json
[
  "-45.0096",
  "-87.2543",
  1579867373
]
dave@howtogeek:~$
```

The output is wrapped in brackets and separated by commas, making it a correctly formed array.

Numeric values can also be manipulated as they're retrieved. Let's pull the timestamp from the ISS position file, and then extract it again and change the value that's returned.

To do so, we type the following:

```
jq ".timestamp" iss.json
```

```
jq ".timestamp - 1570000000" iss.json
```

```
dave@howtogeek:~$ jq ".timestamp" iss.json
1579867373
dave@howtogeek:~$ jq ".timestamp - 1570000000" iss.json
9867373
dave@howtogeek:~$
```

This is useful if you need to add or remove a standard offset from an array of values.

Let's type the following to remind ourselves what the `iss.json` file contains:

```
jq . iss.json
```

```
dave@howtogeek:~$ jq . iss.json
{
  "iss_position": {
    "latitude": "-45.0096",
    "longitude": "-87.2543"
  },
  "message": "success",
  "timestamp": 1579867373
}
dave@howtogeek:~$
```

Let's say we want to get rid of the `message` key:value pair. It doesn't have anything to do with the position of the International Space Station. It's just a flag that indicates the location was retrieved successfully. If it's surplus to requirements, we can dispense with it. (You could also just ignore it.)

We can use `jq`'s delete function, `del()`, to delete a key:value pair. To delete the `message` key:value pair, we type this command:

```
jq "del(.message)" iss.json
```

```
dave@howtogeek:~$ jq "del(.message)" iss.json
{
  "iss_position": {
    "latitude": "-45.0096",
    "longitude": "-87.2543"
  },
  "timestamp": 1579867373
}
dave@howtogeek:~$
```

Note this doesn't actually delete it from the "iss.json" file; it just removes it from the output of the command. If you need to create a new file without the message key:value pair in it, run the command, and then redirect the output into a new file.

## More Complicated JSON Objects

Let's retrieve some more NASA data. This time, we'll use a JSON object that contains [information on meteor impact sites](#) from around the world. This is a bigger file with a far more complicated JSON structure than those we've dealt with previously.

First, we'll type the following to redirect it to a file called "strikes.json":

```
curl -s https://data.nasa.gov/resource/y77d-th95.json | jq
```

```
dave@howtogeek:~$ curl -s https://data.nasa.gov/resource/y77d-th95.json | jq . > strikes.json
```

To see what JSON looks like, we type the following:

```
less strikes.json
```

```
dave@howtogeek:~$ less strikes.json
```

As shown below, the file begins with an opening bracket ([), so the entire object is an array. The objects in the array are collections of key:value pairs, and there's a nested object called geolocation. The geolocation object contains further key:value pairs, and an array called coordinates.

```
[
  {
    "name": "Aachen",
    "id": "1",
    "nametype": "Valid",
    "recclass": "L5",
    "mass": "21",
    "fall": "Fell",
    "year": "1880-01-01T00:00:00.000",
    "reclat": "50.775000",
    "reclong": "6.083330",
    "geolocation": {
      "type": "Point",
      "coordinates": [
        6.08333,
        50.775
      ]
    }
  },
  {
    "name": "Tirupati",
    "id": "2",
    "nametype": "Valid",
    "recclass": "L5",
    "mass": "21",
    "fall": "Fell",
    "year": "1880-01-01T00:00:00.000",
    "reclat": "13.617611",
    "reclong": "79.410777",
    "geolocation": {
      "type": "Point",
      "coordinates": [
        79.410777,
        13.617611
      ]
    }
  },
  {
    "name": "Tissint",
    "id": "3",
    "nametype": "Valid",
    "recclass": "L5",
    "mass": "21",
    "fall": "Fell",
    "year": "1880-01-01T00:00:00.000",
    "reclat": "46.854444",
    "reclong": "7.083333",
    "geolocation": {
      "type": "Point",
      "coordinates": [
        7.083333,
        46.854444
      ]
    }
  },
  {
    "name": "Tjabe",
    "id": "4",
    "nametype": "Valid",
    "recclass": "L5",
    "mass": "21",
    "fall": "Fell",
    "year": "1880-01-01T00:00:00.000",
    "reclat": "34.583333",
    "reclong": "6.083333",
    "geolocation": {
      "type": "Point",
      "coordinates": [
        6.083333,
        34.583333
      ]
    }
  },
  {
    "name": "Tjerebon",
    "id": "5",
    "nametype": "Valid",
    "recclass": "L5",
    "mass": "21",
    "fall": "Fell",
    "year": "1880-01-01T00:00:00.000",
    "reclat": "6.083333",
    "reclong": "101.666667",
    "geolocation": {
      "type": "Point",
      "coordinates": [
        101.666667,
        6.083333
      ]
    }
  },
  {
    "name": "Tomakovka",
    "id": "6",
    "nametype": "Valid",
    "recclass": "L5",
    "mass": "21",
    "fall": "Fell",
    "year": "1880-01-01T00:00:00.000",
    "reclat": "50.775000",
    "reclong": "6.083330",
    "geolocation": {
      "type": "Point",
      "coordinates": [
        6.083330,
        50.775000
      ]
    }
  }
]
}
{
  "strikes.json"
```

Let's retrieve the names of the meteor strikes from the object at index position 995 through the end of the array.

We'll type the following to pipe the JSON through three filters:

```
jq ".[995:] | .[] | .name" strikes.json
```

```
dave@howtogeek:~$ jq ".[995:] | .[] | .name" strikes.json
"Tirupati"
"Tissint"
"Tjabe"
"Tjerebon"
"Tomakovka"
dave@howtogeek:~$
```

The filters function in the following ways:

- `.[995:]`: This tells jq to process the objects from array index 995 through the end of the array. No number after the colon ( : ) is what tells jq to continue to the end of the array.
- `.[]`: This array iterator tells jq to process each object in the array.
- `.name`: This filter extracts the name value.

With a slight change, we can extract the last 10 objects from the array. A "-10" instructs jq to start processing objects 10 back from the end of the array.

We type the following:

```
jq "[-10:] | .[] | .name" strikes.json
```

```
dave@howtogeek:~$ jq "[-10:] | .[] | .name" strikes.json
"Tianzhang"
"Tieschitz"
"Tilden"
"Tillaberi"
"Timochin"
"Tirupati"
"Tissint"
"Tjabe"
"Tjerebon"
"Tomakovka"
dave@howtogeek:~$
```

Just as we have in previous examples, we can type the following to select a single object:

```
jq "[650].name" strikes.json
```

```
dave@howtogeek:~$ jq "[650].name" strikes.json
"Morávka"
dave@howtogeek:~$
```

We can also apply slicing to strings. To do so, we'll type the following to request the first four characters of the name of the object at array index 234:

```
jq "[234].name[0:4]" strikes.json
```

```
dave@howtogeek:~$ jq "[234].name[0:4]" strikes.json
"Derg"
dave@howtogeek:~$
```

We can also see a specific object in its entirety. To do this, we type the following and include an array index without any key:value filters:

```
jq "[234]" strikes.json
```

```
dave@howtogeek:~$ jq ".[234]" strikes.json
{
  "name": "Dergaon",
  "id": "6664",
  "nametype": "Valid",
  "recclass": "H5",
  "mass": "12500",
  "fall": "Fell",
  "year": "2001-01-01T00:00:00.000",
  "reclat": "26.683330",
  "reclong": "93.866670",
  "geolocation": {
    "type": "Point",
    "coordinates": [
      93.86667,
      26.68333
    ]
  }
}
```

If you want to see only the values, you can do the same thing without the key names.

For our example, we type this command:

```
jq ".[234][]" strikes.json
```

```
dave@howtogeek:~$ jq ".[234][]" strikes.json
"Dergaon"
"6664"
"Valid"
"H5"
"12500"
"Fell"
"2001-01-01T00:00:00.000"
"26.683330"
"93.866670"
{
  "type": "Point",
  "coordinates": [
    93.86667,
    26.68333
  ]
}
```

To retrieve multiple values from each object, we separate them with commas in the following command:

```
jq ".[450:455] | .[] | .name, .mass" strikes.json
```

```
dave@howtogeek:~$ jq ".[450:455] | .[] | .name, .mass" strikes.json
"Kaptal-Aryk"
"3500"
"Karakol"
"3000"
"Karatu"
"2220"
"Karewar"
"180"
"Karkh"
"22000"
dave@howtogeek:~$
```

If you want to retrieve nested values, you have to identify the objects that form the “path” to them.

For example, to reference the coordinates values, we have to include the all-encompassing array, the geolocation nested object, and the nested coordinates array, as shown below.

```
[
  {
    "name": "Aachen",
    "id": "1",
    "nametype": "Valid",
    "recclass": "L5",
    "mass": "21",
    "fall": "Fell",
    "year": "1880-01-01T00:00:00.000",
    "reclat": "50.775000",
    "reclong": "6.083330",
    "geolocation": {
      "type": "Point",
      "coordinates": [
        6.08333,
        50.775
      ]
    }
  },
  {
    strikes.json
```

To see the coordinates values for the object at index position 121 of the array, we type the following command:

```
jq ".[121].geolocation.coordinates[]" strikes.json
```

```
dave@howtogeek:~$ jq ".[121].geolocation.coordinates[]" strikes.json
25.8
60.4
dave@howtogeek:~$
```

## The length Function



The `jq` `length` function gives different metrics according to what it's been applied, such as:

- **Strings:** The length of the string in bytes.
- **Objects:** The number of key:value pairs in the object.
- **Arrays:** The number of array elements in the array.

The following command returns the length of the `name` value in 10 of the objects in the JSON array, starting at index position 100:

```
jq ".[100:110] | .[].name | length" strikes.json
```

```
dave@howtogeek:~$ jq ".[100:110] | .[].name | length" strikes.json
6
6
6
13
8
9
6
7
6
6
dave@howtogeek:~$
```

To see how many key:value pairs are in the first object in the array, we type this command:

```
jq ".[0] | length" strikes.json
```

```
dave@howtogeek:~$ jq ".[0] | length" strikes.json
10
dave@howtogeek:~$
```

## The keys Function

You can use the `keys` function to find out about the JSON you've got to work with. It can tell you what the names of the keys are, and how many objects there are in an array.

To find the keys in the `people` object in the "astro.json" file, we type this command:

```
jq ".people.[0] | keys" astro.json
```

```
dave@howtogeek:~$ jq ".people[0] | keys" astro.json
[
  "craft",
  "name"
]
dave@howtogeek:~$
```

To see how many elements are in the `people` array, we type this command:

```
jq ".people | keys" astro.json
```

```
dave@howtogeek:~$ jq ".people | keys" astro.json
[
  0,
  1,
  2,
  3,
  4,
  5
]
dave@howtogeek:~$
```

This shows there are six, zero-offset array elements, numbered zero to five.

## The `has()` Function

You can use the `has()` function to interrogate the JSON and see whether an object has a particular key name. Note the key name must be wrapped in quotation marks. We'll wrap the filter command in single quotes (`'`), as follows:

```
jq '.[ ] | has("nametype")' strikes.json
```

```
dave@howtogeek:~$ jq '.[ ] | has("nametype")' strikes.json
```

Each object in the array is checked, as shown below.

```
true
true
true
true
true
true
true
true
true
true
true
true
true
true
true
true
true
true
true
true
true
dave@howtogeek:~$
```

If you want to check a specific object, you include its index position in the array filter, as follows:

```
jq '.[678] | has("nametype")' strikes.json
```

```
dave@howtogeek:~$ jq '.[678] | has("nametype")' strikes.json
true
dave@howtogeek:~$
```

## Don't Go Near JSON Without It

The `jq` utility is the perfect example of the professional, powerful, fast software that makes living in the Linux world such a pleasure.

This was just a brief introduction to the common functions of this command—there's a whole lot more to it. Be sure to check out the comprehensive [jq manual](#) if you want to dig deeper.