Flask from Scratch...

The basics of web application development with the Flask framework

Up and Running

- Step 1: Install Flask in a virtual environment

```
$ virtualenv venv
$ source venv/bin/activate
(venv) $ pip install flask
```

Up and Running (cont'd)

- Step 2: Create an application instance
  - Save in 'hello_flask.py'

```
from flask import Flask
app = Flask(__name__)
```

Up and Running (cont'd)

- Step 3: Define routes

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def index():
   return '<h1>Hello World!</h1>'

@app.route('/user/<name>')
def user(name):
   return '<h1>Hello, {0}!</h1>'.format(name)
```

Up and Running (cont'd)

- Step 4: Start the development web server

```python
from flask import Flask
app = Flask(__name__)

@app.route('/')
def index():
    return '<h1>Hello World!</h1>'

@app.route('/user/<name>')
def user(name):
    return '<h1>Hello, {0}!</h1>'.format(name)

if __name__ == '__main__':
    app.run(debug=True)
```

Up and Running (cont'd)

- Step 5: Run as a normal Python script!

```
(venv) $ python hello.py
* Running on http://127.0.0.1:5000/
* Restarting with reloader
```

Decorators

- Decorators are used extensively by the framework and extensions to register application provided functions as callbacks.
- Useful decorators:
  - **route** registers functions to handle routes.
  - **before_request** registers a function to run before request handlers.
  - **before_first_request** is similar, but only once at the start.
  - **after_request** registers a function to run after request handlers run.
  - **teardown_request** registers a function to run after request handlers run, even if they throw an exception.
  - **errorhandler** defines a custom error handler.
- Many Flask extensions define their own decorators as well.

Context Globals

- Context globals avoid the need to pass important variables to request handlers.
- Flask's application context defines the following context globals:
  - **current_app** is the application instance.
  - **g** is a global dictionary for request data storage.
- Flask's request context defines the following context globals:
  - **request** is the request being processed by the thread.
  - **session** is the user session storage.

Helper functions

- Flask provides several auxiliary functions:
  - **url_for()** generates links to routes or static files.
  - **render_template()** renders Jinja2 templates.
  - **redirect()** generates a redirect response.
  - **jsonify()** generates a JSON response.
  - **abort()** generates an error response (throws an exception).
  - **flash()** registers a message to display to the user.

Converters

```
from flask import Flask
app = Flask(__name__)

@app.route('/blog/<int:postID>')
def show_blog(postID):
   return 'Blog Number %d' % postID

@app.route('/rev/<float:revNo>')
def revision(revNo):
   return 'Revision Number %f' % revNo

if __name__ == '__main__':
   app.run(debug=True)
```

Routing - caveats

```
from flask import Flask
app = Flask(__name__)

@app.route('/flask')
def hello_flask():
   return 'Hello Flask'

@app.route('/python/')
def hello_python():
   return 'Hello Python'

if __name__ == '__main__':
   app.run(debug=True)
```

Both the rules appear similar but in the second rule, trailing slash (/) is used. As a result, it becomes a canonical URL. Hence, using /python or /python/ returns the same output. However, in case of the first rule, /flask/ URL results in 404 Not Found page.

Flask – URL Building

```
from flask import Flask, redirect, url_for
app = Flask(__name__)

@app.route('/admin')
def hello_admin():
   return 'Hello Admin'

@app.route('/guest/<guest>')
def hello_guest(guest):
   return 'Hello %s as Guest' % guest

@app.route('/user/<name>')
def hello_user(name):
   if name =='admin':
      return redirect(url_for('hello_admin'))
   else:
      return redirect(url_for('hello_guest',guest = name))

if __name__ == '__main__':
   app.run(debug = True)
```

Flask – HTTP methods

GET	Sends data in unencrypted form to the server. Most common method.
HEAD	Same as GET, but without response body
POST	Used to send HTML form data to server. Data received by POST method is not cached by server.
PUT	Replaces all current representations of the target resource with the uploaded content.
DELETE	Removes all current representations of the target resource given by a URL

By default, the Flask route responds to the GET requests. However, this preference can be altered by providing methods argument to route() decorator.

POST – login.html

```
<html>
   <body>

      <form action = "http://localhost:5000/login" method = 
"post">
         <p>Enter Name:</p>
         <p><input type = "text" name = "nm" /></p>
         <p><input type = "submit" value = "submit" /></p>
      </form>

   </body>
</html>
```

POST – flask_post.py

```
from flask import Flask, redirect, url_for, request
app = Flask(__name__)

@app.route('/success/<name>')
def success(name):
   return 'welcome %s' % name

@app.route('/login',methods = ['POST', 'GET'])
def login():
   if request.method == 'POST':
      user = request.form['nm']
      return redirect(url_for('success',name = user))
   else:
      user = request.args.get('nm')
      return redirect(url_for('success',name = user))

if __name__ == '__main__':
   app.run(debug = True)
```

Flask – Templates

Instead of returning hardcode HTML from the function, a HTML file can be rendered by the render_template() function.

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def index():
   return render_template(‘hello.html’)

if __name__ == '__main__':
   app.run(debug = True)
```

Flask will try to find the HTML file in the templates folder, in the same folder in which this script is present.

- Application folder
  - Hello.py
  - templates
    - hello.html

Flask – the jinja2 templates engine

Flask uses jinja2 template engine. A web template contains HTML syntax interspersed placeholders for variables and expressions (in these case Python expressions) which are replaced values when the template is rendered.

The following code is saved as hello.html in the templates folder.

```
<!doctype html>
<html>
   <body>

      <h1>Hello {{ name }}!</h1>

   </body>
</html>
```

```
from flask import Flask, render_template
app = Flask(__name__)

@app.route('/hello/<user>')
def hello_name(user):
   return render_template('hello.html', name = user)

if __name__ == '__main__':
   app.run(debug = True)
```

The jinja2 template - delimiters

The jinja2 template engine uses the following delimiters for escaping from HTML:

- {% ... %} for Statements
- {{ ... }} for Expressions to print to the template output
- {# ... #} for Comments not included in the template output
- # ... ## for Line Statements

Jinja2 – conditional statement

The URL rule to the hello() function accepts the integer parameter. It is passed to the hello.html template. Inside it, the value of number received (marks) is compared (greater or less than 50) and accordingly HTML is conditionally rendered.

The Python Script is as follows −

```
from flask import Flask, render_template
app = Flask(__name__)

@app.route('/hello/<int:score>')
def hello_name(score):
   return render_template('hello2.html', marks = score)

if __name__ == '__main__':
   app.run(debug = True)
```

HTML template script of hello2.html is as follows −

```
<!doctype html>
<html>
   <body>

      {% if marks>50 %}
      <h1> Your result is pass!</h1>
      {% else %}
      <h1>Your result is fail</h1>
      {% endif %}

   </body>
</html>
```

Jinja2 – template loop constructs

The Python loop constructs can also be employed inside the template. In the following script, the result() function sends a dictionary object to template results.html when URL http://localhost:5000/result is opened in the browser.

```
from flask import Flask, render_template
app = Flask(__name__)

@app.route('/result')
def result():
   dict = {'phy':50,'che':60,'maths':70}
   return render_template('result.html', result = dict)

if __name__ == '__main__':
   app.run(debug = True)
```

Save the following HTML script as result.html in the templates folder.

```
<!doctype html>
<html>
   <body>

      <table border = 1>
         {% for key, value in result.items() %}

            <tr>
               <th> {{ key }} </th>
               <td> {{ value }} </td>
            </tr>

         {% endfor %}
      </table>

   </body>
</html>
```

Flask – Static Files

A web application often requires a static file such as a javascript file or a CSS file supporting the display of a web page. Usually, the web server is configured to serve them for you, but during the development, these files are served from static folder in your package or next to your module and it will be available at /static on the application.

In the following example, a javascript function defined in hello.js is called on OnClick event of HTML button in index.html, which is rendered on ‘/’ URL of the Flask application.

```
from flask import Flask, render_template
app = Flask(__name__)

@app.route("/")
def index():
   return render_template("index.html")

if __name__ == '__main__':
   app.run(debug = True)
```

hello.js contains sayHello() function.

```
function sayHello() {
   alert("Hello World")
}
```

The HTML script of index.html is given below.

```
<html>

   <head>
      <script type = "text/javascript"
         src = "{{ url_for('static', filename = 'hello.js') }}" ></script>
   </head>

   <body>
      <input type = "button" onclick = "sayHello()" value = "Say Hello" />
   </body>

</html>
```

Flask – Request Object

The data from a client’s web page is sent to the server as a global request object. In order to process the request data, it should be imported from the Flask module.

Important attributes of request object are listed below −

Form	It is a dictionary object containing key and value pairs of form parameters and their values.
args	parsed contents of query string which is part of URL after question mark (?).
Cookies	dictionary object holding Cookie names and values.
files	data pertaining to uploaded file.
method	current request method.

Flask – Sending Form Data to Template

We have already seen that the http method can be specified in URL rule. The Form data received by the triggered function can collect it in the form of a dictionary object and forward it to a template to render it on a corresponding web page.

In the following example, ‘/’ URL renders a web page (student.html) which has a form. The data filled in it is posted to the ‘/result’ URL which triggers the result() function.

The results() function collects form data present in request.form in a dictionary object and sends it for rendering to result.html.

The template dynamically renders an HTML table of form data.

Flask – Sending Form Data to Template

```python
from flask import Flask, render_template, request
app = Flask(__name__)

@app.route('/')
def student():
   return render_template('student.html')

@app.route('/result',methods = ['POST', 'GET'])
def result():
   if request.method == 'POST':
      result = request.form
      return render_template("result_request.html",result = result)

if __name__ == '__main__':
   app.run(debug = True)
```

Given below is the HTML script of student.html.

```html
<html>
   <body>

      <form action = "http://localhost:5000/result" method = "POST">
         <p>Name <input type = "text" name = "Name" /></p>
         <p>Physics <input type = "text" name = "Physics" /></p>
         <p>Chemistry <input type = "text" name = "chemistry" /></p>
         <p>Maths <input type ="text" name = "Mathematics" /></p>
         <p><input type = "submit" value = "submit" /></p>
      </form>

   </body>
</html>
```

Code of template (result_request.html):

```html
<!doctype html>
<html>
   <body>

      <table border = 1>
         {% for key, value in result.items() %}

            <tr>
               <th> {{ key }} </th>
               <td> {{ value }} </td>
            </tr>

         {% endfor %}
      </table>

   </body>
</html>
```

Creating Line Charts in D3.js

- What is D3.js?
  - a JavaScript library for manipulating documents based on data in an interactive way. It makes use of HTML5, JavaScript, SVG and CSS3.

- D3 source can be downloaded from: http://d3js.org/

Creating the X and Y Axes

SVG is needed in our HTML page. SVG is a XML based vector image format that offers support for interaction and animation.

```
<html>
  <head>
    <script src="jquery.js"></script>
    <script src="d3.v3.js"></script>
    <script>
      $(function() {
        InitChart();
      });
      function InitChart() {
        // Chart creation code goes here
      }
    </script>
  </head>
  <body>
    <svg id="svgVisualize" width="500" height="500"></svg>
  </body>
</html>
```

Drawing the X and Y Axes (1)

- d3.scale.linear():
  - used to create a quantitative scale.
  - defines the **range** and **domain** of each axis.
  - The **domain** defines the minimum and maximum values displayed on the graph, while the **range** is the amount of the SVG we'll be covering.
    - Our svg is 500×500 so, let's define our range as 40×400.

```
var xRange = d3.scale.linear().range([40, 400]).domain([0,100]);
var yRange = d3.scale.linear().range([40, 400]).domain([0,100]);
```

Drawing the X and Y Axes (2)

- axis.scale(): to scale the axes.

```
var xAxis = d3.svg.axis().scale(xRange);
var yAxis = d3.svg.axis().scale(yRange);
```

- Append the x and y axes to the SVG element:

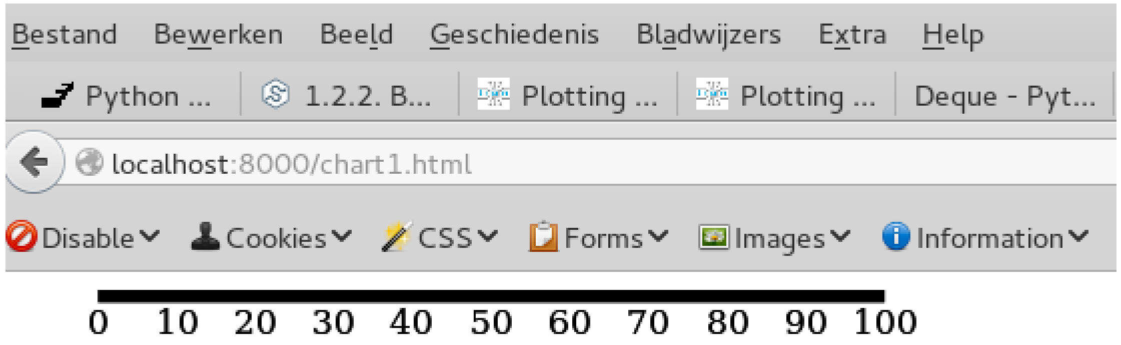
```
vis.append("svg:g").call(xAxis);
vis.append("svg:g").call(yAxis);
```

InitChart()

```
function InitChart() {
  var vis = d3.select("#svgVisualize");
  var xRange = d3.scale.linear().range([40, 400]).domain([0,100]);
  var yRange = d3.scale.linear().range([40, 400]).domain([0,100]);
  var xAxis = d3.svg.axis().scale(xRange);
  var yAxis = d3.svg.axis().scale(yRange);
  vis.append("svg:g").call(xAxis);
  vis.append("svg:g").call(yAxis);
}
```

Demo

```
> python -m SimpleHTTPServer 8000
Serving HTTP on 0.0.0.0 port 8000 ...
127.0.0.1 - - [29/Oct/2015 19:20:39] "GET /chart1.html HTTP/1.1" 200 -
127.0.0.1 - - [29/Oct/2015 19:20:39] "GET /jquery-1.8.0.min.js HTTP/1.1"
200 -
```

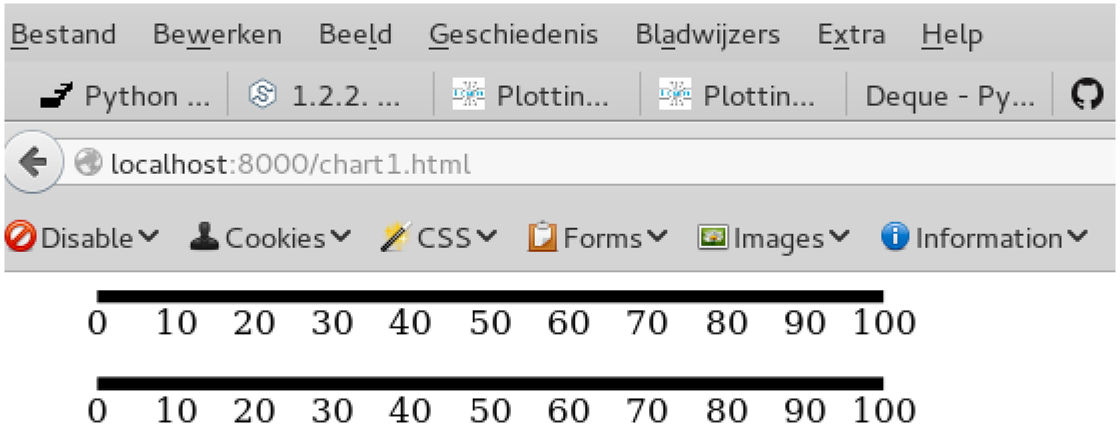


Demo

- There are two lines overlapping each other. To separate the axes, modify the code where we appended the y-axis as shown below:

```
vis.append("svg:g").call(yAxis).attr("transform", "translate(0,40)");
```

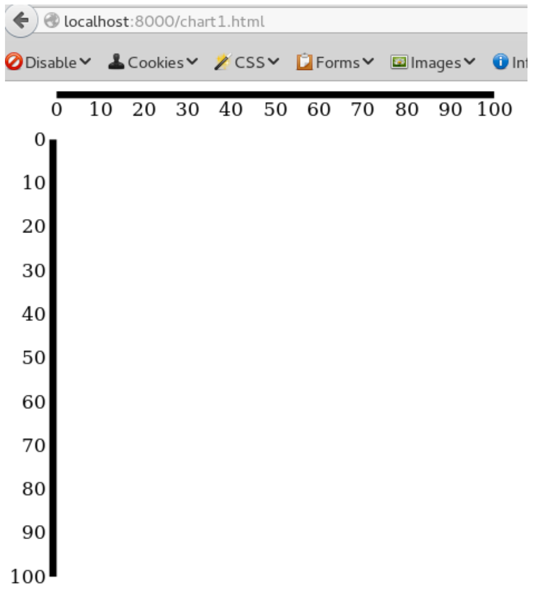
- Now, you can see both axes because we have moved our y-axis by 40 units.



Demo

1. move the y-axis 40 units from the x-axis and 0 units from the y-axis
2. change its orientation to left.

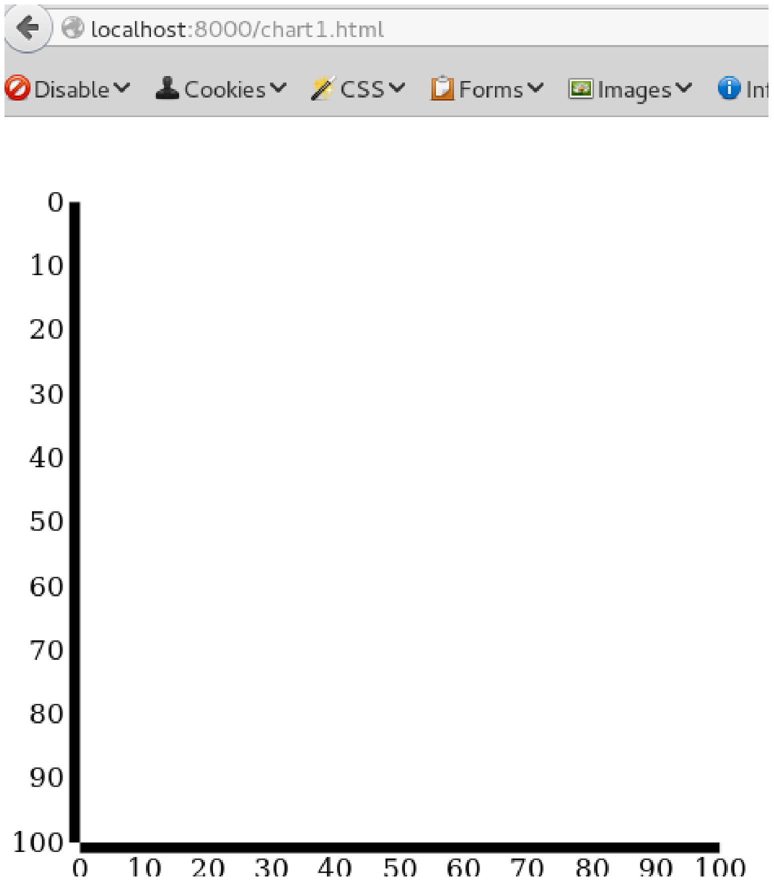
```
function InitChart() {
  var vis = d3.select("#svgVisualize");
  var xRange = d3.scale.linear().range([40, 400]).domain([0,100]);
  var yRange = d3.scale.linear().range([40, 400]).domain([0,100]);
  var xAxis = d3.svg.axis().scale(xRange);
  var yAxis = d3.svg.axis().scale(yRange).orient("left");
  vis.append("svg:g").call(xAxis);
  vis.append("svg:g").call(yAxis).attr("transform", "translate(40,0)");
}
```



Demo

- the x-axis needs to be moved down: use transform

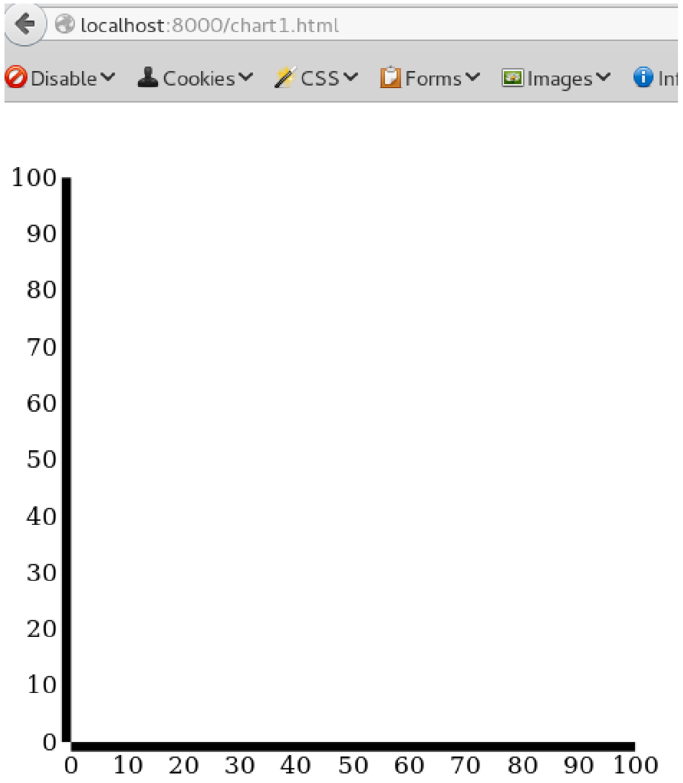
```
vis.append("svg:g").call(xAxis).attr("transform", "translate(0,400)");
```



Demo

- The y-axis scale goes from 100 to 0. We need to invert it like so:

```
var yRange = d3.scale.linear().range([400, 40]).domain([0,100]);
```



Creating Line Charts

- Data:

```
var lineData = [{
  x: 1,
  y: 5
}, {
  x: 20,
  y: 20
}, {
  x: 40,
  y: 10
}, {
  x: 60,
  y: 40
}, {
  x: 80,
  y: 5
}, {
  x: 100,
  y: 60
}];
```

- Needed: an<svg> element to plot our graph on.

```
<svg id="visualisation" width="1000" height="500"></svg>
```

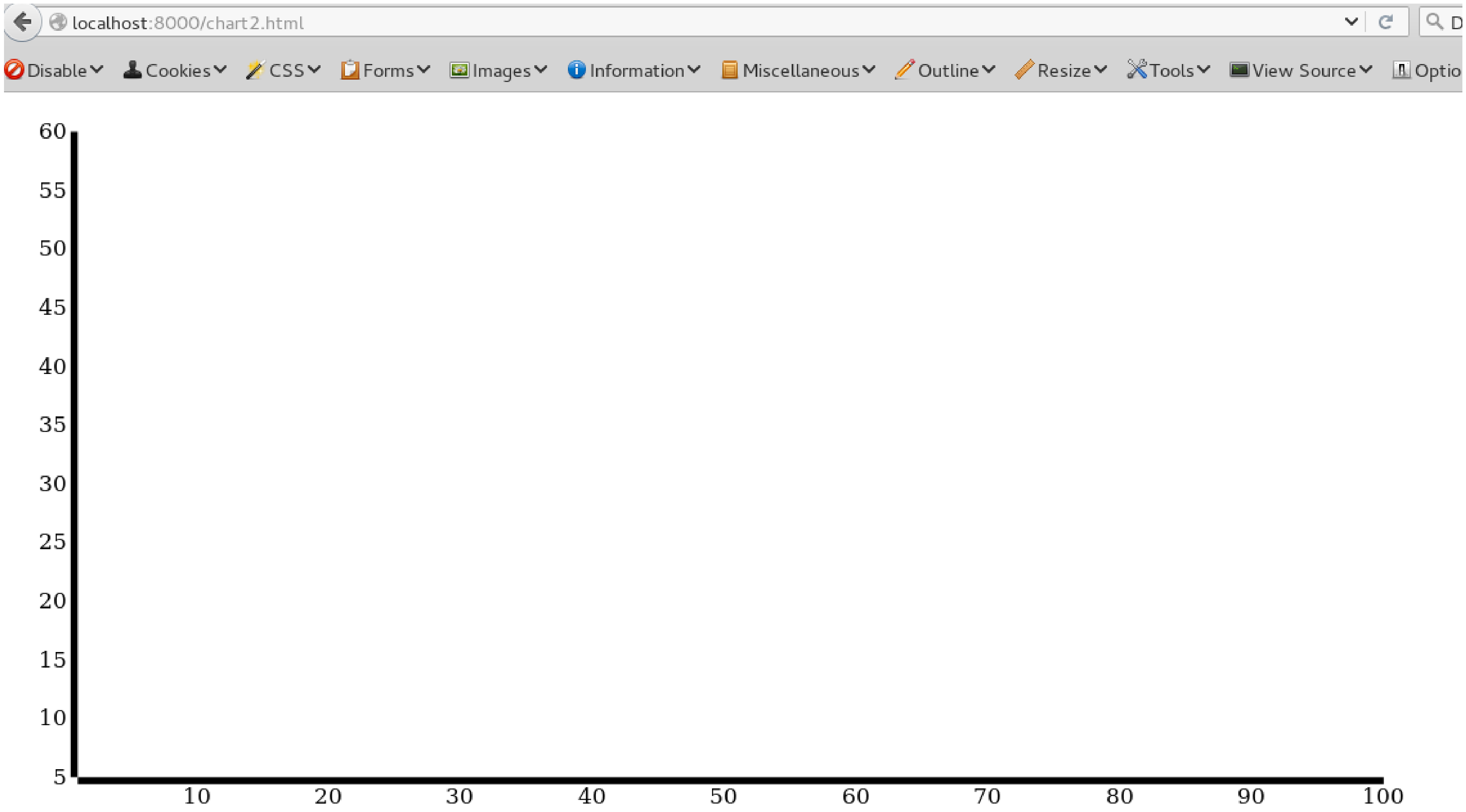
Creating Line Charts

- create the x and y axes
  - declare a domain and range.
  - The domain defines the minimum and maximum values displayed on the graph, while the range is the amount of the SVG we’ll be covering.

Creating Line Charts

```
var vis = d3.select('#visualisation'),
    WIDTH = 1000,
    HEIGHT = 500,
    MARGINS = {
      top: 20,
      right: 20,
      bottom: 20,
      left: 50
    },
    xRange = d3.scale.linear().range([MARGINS.left, WIDTH - MARGINS.right]).domain([d3.min(lineData, function(d) {
      return d.x;
    }), d3.max(lineData, function(d) {
      return d.x;
    })]),
    yRange = d3.scale.linear().range([HEIGHT - MARGINS.top, MARGINS.bottom]).domain([d3.min(lineData, function(d) {
      return d.y;
    }), d3.max(lineData, function(d) {
      return d.y;
    })]),
    xAxis = d3.svg.axis()
      .scale(xRange)
      .tickSize(5)
      .tickSubdivide(true),
    yAxis = d3.svg.axis()
      .scale(yRange)
      .tickSize(5)
      .orient('left')
      .tickSubdivide(true);
vis.append('svg:g')
  .attr('class', 'x axis')
  .attr('transform', 'translate(0,' + (HEIGHT - MARGINS.bottom) + ')')
  .call(xAxis);
vis.append('svg:g')
  .attr('class', 'y axis')
  .attr('transform', 'translate(' + (MARGINS.left) + ',0)')
  .call(yAxis);
```

Creating Line Charts - Demo



Plotting the Line

- Apply the xRange and the yRange to the coordinates to transform them into the plotting space and to draw a line across the plotting space.
  - d3.svg.line() is used to draw our line graph.
    - create a line generator function which returns the x and y coordinates from our data to plot the line.

```
var lineFunc = d3.svg.line()
  .x(function(d) {
    return xRange(d.x);
  })
  .y(function(d) {
    return yRange(d.y);
  })
  .interpolate('linear');
```

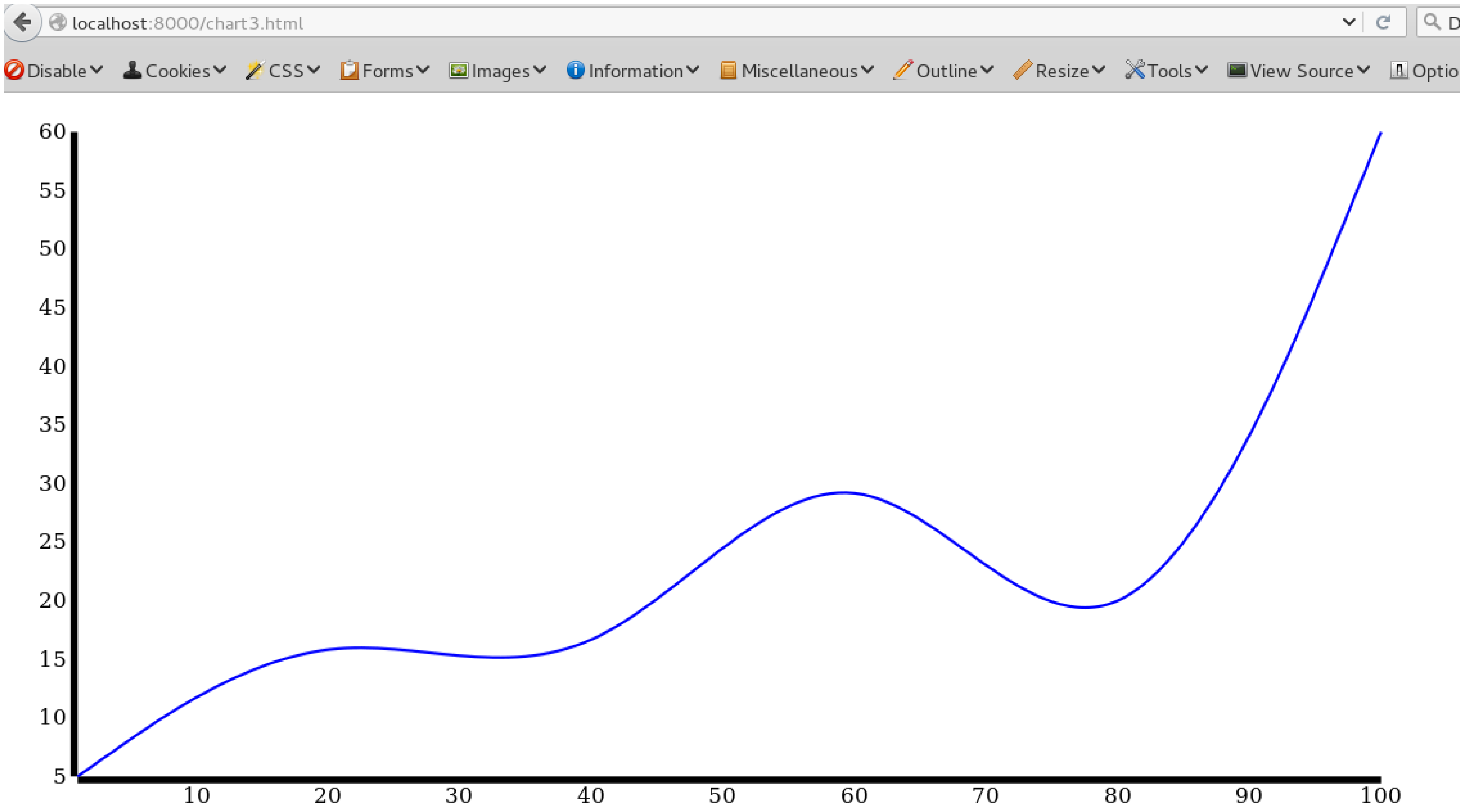
Plotting the Line

- Next, we need to set the d attribute of the SVG path to the coordinates returned from the line function. This is accomplished using the following code:

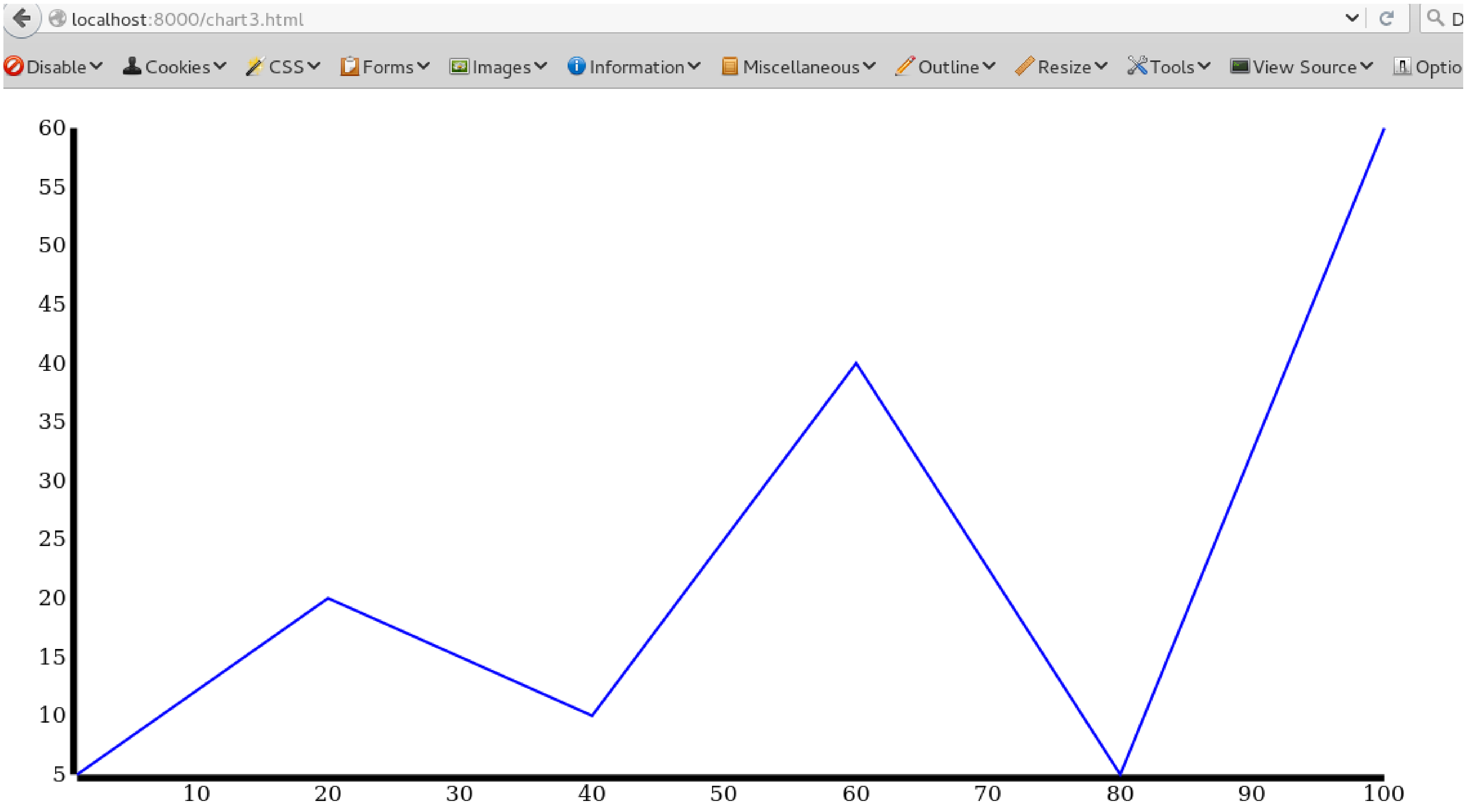
```
vis.append('svg:path')
  .attr('d',
lineFunc(lineData))
  .attr('stroke', 'blue')
  .attr('stroke-width', 2)
  .attr('fill', 'none');
```

The line color is set using 'stroke'. The line’s width is defined using 'stroke-width'. 'fill' is set to none, as not to fill the graph boundaries.

Plotting the Line Chart – 'basis' interpolation



Plotting the Line Chart – 'linear' interpolation


localhost:8000/chart3.html
Disable
Cookies
CSS
Forms
Images
Information
Miscellaneous
Outline
Resize
Tools
View Source
Optio