Graphing with chartist

Using only HTML, javascript, and python to create visualizations on the web.

https://github.com/C3NZ/visualizing-trends

By Vincenzo Marcella

Graphing with Chartist

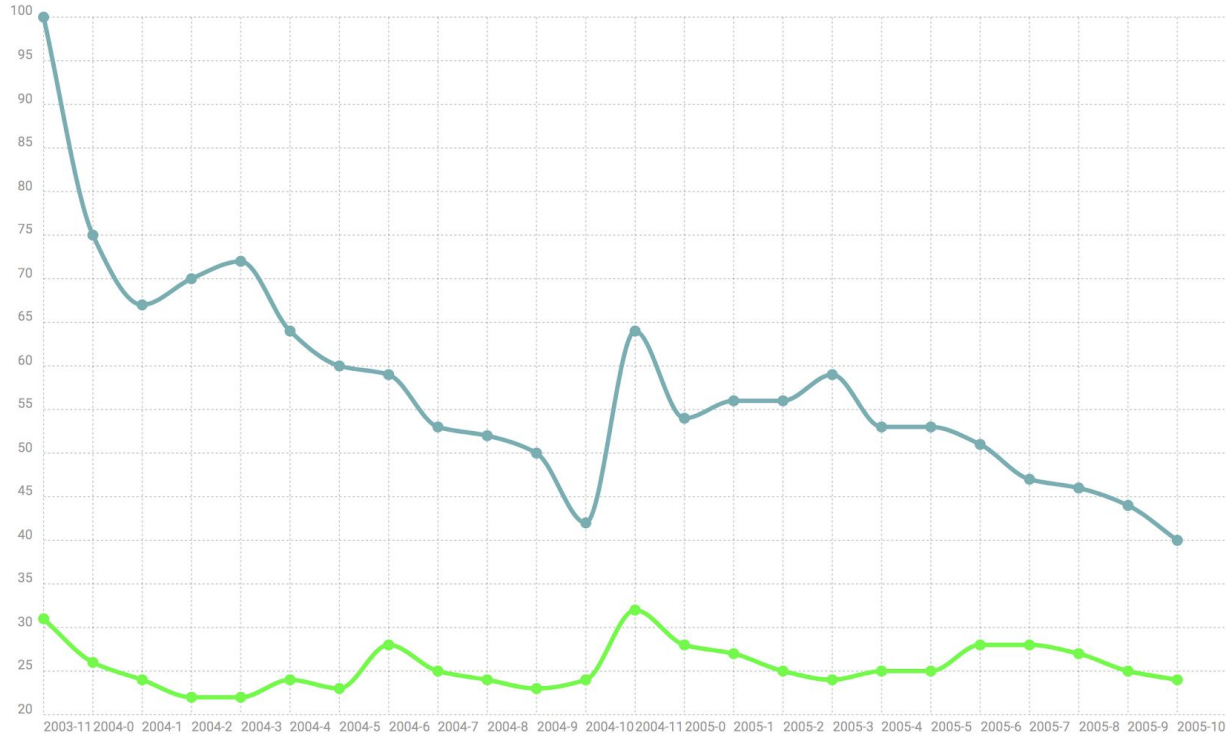
Time Frame

2004 to 2005

Selected columns

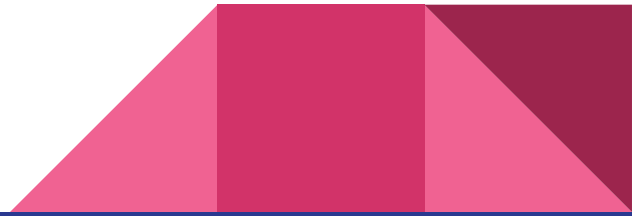
- ✓ Diet
- ✓ Gym
- Finance

Update chart



Prerequisites

- Backend dependencies
  - Flask - Web server
  - Pandas - Data processing
  - virtualenv - python environment isolation (optional)
- Frontend dependencies
  - Chartist.js
  - Axios
- Etc.
  - The csv containing the data we need
  - Docker & Docker compose (Ease of setup)



Getting started

Inside of a new directory, we're going to need:

- ***app.py***
- ***templates*** folder containing:
  - index.html
- ***static*** folder container:
  - index.js
  - index.css (we're not going to focus on styling choices made)
- ***multiTimeline.csv***

app.py

Our app.py is the backend portion of the application. The first things that need to be imported are the

- pandas - For data processing
- flask - For web hosting

We setup our flask application and then attach df, the pandas dataframe, to the application.

```
# Attach our dataframe to our app
app.loaded_csv = pd.read_csv("multiTimeline.csv", skiprows=1)
app.loaded_csv.columns = ["month", "diet", "gym", "finance"]

# Convert dates to datetimes
app.loaded_csv["month"] = pd.to_datetime(
    app.loaded_csv["month"], format="%Y-%d")
```

get_root

Inside of our **app.py**, we define a get_root method for obtaining the index.html file we had specified earlier.

@app.route(...) - Function decarator that converts our function get_root into the route handler for the route and methods we specify

render_template - will render index.html (currently empty)

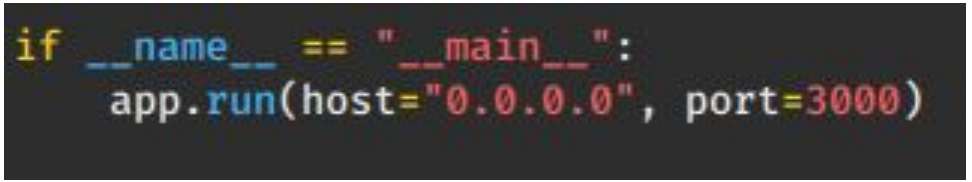
200 - the http response status code

```
@app.route("/", methods=["GET"])
def get_root():
    """
        Root route that returns the index page
    """
    return render_template("index.html"), 200
```

Running our server

In order to run our server, we need to add two more lines of code to the bottom of our.

app.run() - tells the flask server to start. The options that are passed in, host and port, tell flask the host ip address and port of that host to run on.



Setting up our frontend

The head and body of the beast

Inside of the header we simply just set the title of the page and grab the styling sheets we will need for our web page.

```
<head>
    <title>Graphing a time series</title>
    <link rel="stylesheet" href="//cdn.jsdelivr.net/chartist.js/latest/chartist.min.css">
    <link rel="stylesheet" href="https://fonts.googleapis.com/css?family=Roboto">
    <link rel="stylesheet" href="/static/index.css">
</head>
```

Breaking it down

First, we setup a div called container that will wrap our entire application. Inside we also create another div that will be used for wrapping the “data area” of the application

```
<div class="container">

    <h1>Graphing with Chartist</h1>
    <div class="data-area">
    </div>

</div>
```

Controlling our data

This next div, control, will allow us to control the graph itself.

The ***control-date*** div is what we will use to control the time frame that is selected.

The ***control-column*** div is what we will use to control the columns selected from the dataframe.

The ***control-button*** div is what we will use to update our chart.

```
<div class="control">
  <!-- Range controllers -->
  <h3>Time Frame</h3>
  <div class="control-date">
    <input class="underlined-input" type="number" value="2004" min="2004" max="2017" />
    <p>to</p>
    <input class="underlined-input" type="number" value="2005" min="2004" max="2017" />
  </div>


  <!-- Trend controllers -->
  <h3>Selected columns</h3>
  <div class="control-column">
    <ul>
      <li class="control-column-item checked">
        Diet <input type="color" value="#0140BA"/>
      </li>
      <li class="control-column-item checked">
        Gym <input type="color" value="#01FF04"/>
      </li>
      <li class="control-column-item">
          Finance <input type="color" value="
      </li>
    </ul>
  </div>

  <!-- Update the chart -->
  <div class="control-button">
    <input type="button" onclick="updateChart()" value="Update chart">
  </div>
</div>
```

Charting it

This last div will be the chart itself.

The outter div labelled by the class ***chart*** is used to wrap the div that will actually be drawing the chart.

ct-chart - is a class provided by Chartist that is used for rendering charts

ct-perfect-fourth - is another class provided by chartist used for maintaining the 4:3 aspect ratio of the chart.

```
<!-- Rendered chart -->
<div class="chart">
  <div class="ct-chart ct-perfect-fourth"></div>
</div>
```

Wrapping up

To wrap up, we add some script tags right before the end of the body to add the functionality

```
<script src="//cdn.jsdelivr.net/chartist.js/latest/chartist.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/axios/0.18.0/axios.js"></script>
<script src="/static/index.js"></script>
```

```
<body>
  <div class="container">
    <h1>Graphing with Chartist</h1>

    <div class="data-area">
      <div class="control">
        <!-- Range controllers -->
        <h3>Time Frame</h3>
        <div class="control-date">
          <input class="underlined-input" type="number" value="2004" min="2004" max="2017" />
          <p>to</p>
          <input class="underlined-input" type="number" value="2005" min="2004" max="2017" />
        </div>


        <!-- Trend controllers -->
        <h3>Selected columns</h3>
        <div class="control-column">
          <ul>
            <li class="control-column-item checked">
              Diet <input type="color" value="#0140BA"/>
            </li>
            <li class="control-column-item checked">
              Gym <input type="color" value="#01FF04"/>
            </li>
            <li class="control-column-item">
                Finance <input type="color" value="#FF0D0A"/>
            </li>
          </ul>
        </div>

        <!-- Update the chart -->
        <div class="control-button">
          <input type="button" onclick="updateChart()" value="Update chart">
        </div>
      </div>

      <!-- Rendered chart -->
      <div class="chart">
        <div class="ct-chart ct-perfect-fourth"></div>
      </div>
    </div>
  </div>
  <script src="//cdn.jsdelivr.net/chartist.js/latest/chartist.min.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/axios/0.18.0/axios.js"></script>
  <script src="/static/index.js"></script>
</body>
```

Sending and receiving data

back to our app.py

Right under our get_root route, we create a new function that will be used to handle sending time series data based on the request we will be sending in the future. This is a lot to look at, so let's break it down.

```python
@app.route("/time_series", methods=["GET"])
def get_time_series_data():
    """
    Return the necessary data to create a time series
    """
    # Grab the requested years and trends from the query arguments
    # default range is from 2004-2005 and default trends are diet and gym.
    range_of_years = [int(year) for year in request.args.getlist("years")]
    trends = request.args.getlist("trends")

    # Generate a list of all the months we need to get
    min_year = min(range_of_years)
    max_year = max(range_of_years)

    # Grab all of the data specified from start to stop range.
    selected_date_range = app.loaded_csv[
        (app.loaded_csv["month"] ⩾ datetime.datetime(min_year, 1, 1)) &
        (app.loaded_csv["month"] ⩽ datetime.datetime(max_year, 12, 31))
    ]

    # Slice the DF to include only the trends we want and then to sort our
    # dataframe by those trends.
    requested_trend_data = selected_date_range[["month"] + trends]
    requested_trend_data = requested_trend_data.sort_values(by=["month"])

    # Return the dataframe as json
    return requested_trend_data.to_json(), 200
```

Parsing the query parameters

Our request is going to be structured like this:

GET http://localhost/time_series?years=2004&years=2005&trends=diet&trends=fitness

As we can see, we have two query params, **years** *and* ***trends***.

years corresponds to the time frame, while trends corresponds to the specific categories.

Since we reuse the query params for multiple values, they're provided as lists in which we need to parse through.

```
# Grab the requested years and trends from the query arguments
# default range is from 2004-2005 and default trends are diet and gym.
range_of_years = [int(year) for year in request.args.getlist("years")]
trends = request.args.getlist("trends")
```

Handling time.

Because we're presented only two dates, we need to obtain the min and max between the two in order to obtain the correct range of dates to return for use of the frontend.

```
# Generate a list of all the months we need to get
min_year = min(range_of_years)
max_year = max(range_of_years)
```

Grabbing the dates we want

To obtain the data that we want to return to the user, we slice the dataframe for all dates that are in the range:

min_year <= x <= max_year

Where x are all the values between the min and max year. This grabs us all of the months from the dataframe and then allows us to slice it further in order to obtain only the requested trends. We then sort all of the data by the month. (Datetime)

```
# Grab all of the data specified from start to stop range.
selected_date_range = app.loaded_csv[
    (app.loaded_csv["month"] ≥ datetime.datetime(min_year, 1, 1)) &
    (app.loaded_csv["month"] ≤ datetime.datetime(max_year, 12, 31))
]

# Slice the DF to include only the trends we want and then to sort our
# dataframe by those trends.
requested_trend_data = selected_date_range[["month"] + trends]
requested_trend_data = requested_trend_data.sort_values(by=["month"])
```

Returning a response

```
# Return the dataframe as json
return requested_trend_data.to_json(), 200
```

Creating a chart and grabbing input.

in the first line, we create our chart using **new Chartist.Line(“.ct-chart”, {})**. This grabs the chart div we specified earlier, and instantiates it with no data.

Next we create an array of lines. These lines will be explained later, but essentially deal with updating our chart.

columnList and **columns** are the inputs for the from our control column.

```
const chart = new Chartist.Line(".ct-chart", {});

const lines = ["a", "b", "c"];

// Grab all the columns
const columnList = document.querySelector(".control-column ul");
const columns = columnList.children;
```

Adding functionality to our input

Using the input column list we previously defined, we add an event listener to it to toggle a “checked” class for signifying that this column was checked. This will be handy both in terms of aesthetic and functional purpose.

```
// Add an event listener to all selectable trends.
columnList.addEventListener("click", event => {
  if (event.target.tagName === "LI") {
    event.target.classList.toggle("checked");
  }
});
```

Grabbing chart data

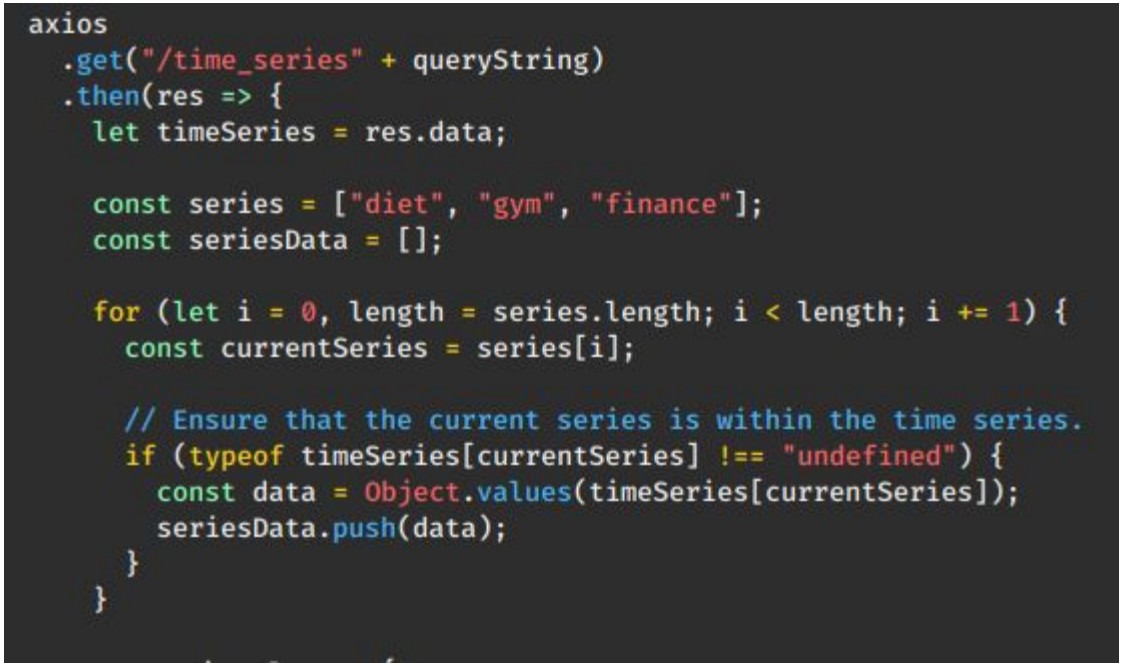
This is a lot to digest, so lets break it down at a higher level first:

1. We use axios to create a get request to our backend + the query string.
2. We create a list of series data from the json data provided to us by our dataframe.
3. setup our chart data object (labels and series data)
4. update the chart and lines drawn to the screen.

```
// Get the initial chart data
function getChartData(queryString = "?years=2004&years=2005&trends=diet&trends=gym") {
  axios
    .get("/time_series" + queryString)
    .then(res => {
      let timeSeries = res.data;

      const series = ["diet", "gym", "finance"];
      const seriesData = [];

      for (let i = 0, length = series.length; i < length; i += 1) {
        const currentSeries = series[i];

        // Ensure that the current series is within the time series.
        if (typeof timeSeries[currentSeries] !== "undefined") {
          const data = Object.values(timeSeries[currentSeries]);
          seriesData.push(data);
        }
      }

      const chartData = {
        labels: Object.values(timeSeries.month).map(datetime => {
          const date = new Date(datetime);
          return `${date.getFullYear()}-${date.getMonth()}`;
        }),
        series: seriesData
      };

      // Update the chart and line colors.
      chart.update(chartData);
      updateLines(columnList.children);
    })
    .catch(err => console.error(err));
}
```

Sending the request and grabbing data

Using the query string passed into the function by default or specification, we create a request to the endpoint we had setup earlier. With the response, we then grab the series Data that we're going to be drawing to the chart if it was returned in the response.



```
axios
  .get("/time_series" + queryString)
  .then(res => {
    let timeSeries = res.data;

    const series = ["diet", "gym", "finance"];
    const seriesData = [];

    for (let i = 0, length = series.length; i < length; i += 1) {
      const currentSeries = series[i];

      // Ensure that the current series is within the time series.
      if (typeof timeSeries[currentSeries] !== "undefined") {
        const data = Object.values(timeSeries[currentSeries]);
        seriesData.push(data);
      }
    }
```

Create chart data

We create the chart data object which is what will be used to update our chart with our time series data.

labels - defines our labels that will be used on the x axis for our points.

series - defines our series data that will be plotted on the y axis

```
const chartData = {
  labels: Object.values(timeSeries.month).map(datetime => {
    const date = new Date(datetime);
    return `${date.getFullYear()}-${date.getMonth()}`;
  }),
  series: seriesData
};

// Update the chart and line colors.
chart.update(chartData);
updateLines(columnList.children);
```

Final catch!

attached to our ***axios.get(...).then(...)*** should be the snippet on the right, looking like this

axios.get(url).then(res…).catch(err…)

(refer back to slide 22 to see the entire function)

```
})
.catch(err => console.error(err));
```

Updating the lines

Updating the lines looks very complicated but is not. All we do is iterate over the columns that are allowed to be selected by the user.

We check if they're selected and if they are, color the line and points of the data being plotted on the chart. We use the lines array that we previously mentioned we would explain. Chartist uses letters for which lines are being colored, so we need to make sure that lines are being colored properly.

```
// Update the lines drawn with the correct colors
function updateLines(items) {
  let lineNumber = 0;
  // Grab all the checked columns
  for (let i = 0, length = items.length; i < length; i += 1) {
    // Only color the checked off lines
    if (items[i].classList.contains("checked")) {
      currentLine =
          document.querySelector(`.ct-series-${lines[lineNumber]} .ct-line`);
      currentPoints =
          document.querySelectorAll(`.ct-series-${lines[lineNumber]} .ct-point`);
      currentLine.style.cssText =
          `stroke: ${items[i].children[0].value} !important;`;

      // Color every point
      for (let j = 0, length = currentPoints.length; j < length; j += 1) {
        currentPoints[j].style.cssText =
            `stroke: ${items[i].children[0].value} !important;`;
      }
      lineNumber += 1;
    }
  }
}
```

Updating the chart

This is the function we use to create the query string for when we need to update the data we're going to be plotting on the chart. We generate the query parameters by going through both the time frames and columns that can be selected through our inputs.

Once the query string has been created, we then call our ***getChartData()*** function.

```
// Updating the chart data
function updateChart() {
  let queryString;

  // Get both input fields from the time frame section
  const timeFrames = document.querySelectorAll(".control-date > input");

  // Grab all time ranges
  for (let i = 0, length = timeFrames.length; i < length; i += 1) {
    time = timeFrames[i].value;

    // Start or add to the query string
    if (typeof queryString === "undefined") {
      queryString = "?years=" + time;
    } else {
      queryString += "&years=" + time;
    }
  }

  let currLine = 0;
  // Grab all the checked columns
  for (let i = 0, length = trends.length; i < length; i += 1) {
    if (trends[i].classList.contains("checked")) {
      queryString += "&trends=" + trends[i].innerText.toLowerCase();
    }
  }

  getChartData(queryString);
}
```

Finishing up

We add event listeners to all of our column color inputs so that the lines get redrawn when the colors are being updated.

To finish it all off, we place a initial ***getChartData()*** function call at the bottom to make our first request.

```
// Grab all the checked columns
for (let i = 0, length = trends.length; i < length; i += 1) {
  if (trends[i].classList.contains("checked")) {
    queryString += "&trends=" + trends[i].innerText.toLowerCase();
  }
}
```

```
getChartData();
```

Closing thoughts

- Improvements
  - handling the json manually (for better output)
- Improving the slideshow

https://github.com/C3NZ/visualizing-trends