# Matplotlib Intro

In this reading, we'll learn how to create plots from Pandas data. Pandas uses a module called matplotlib to create plots. The matplotlib library is designed to resemble MATPLOT (a programming language for matrices and environment that support visualization).

While we could import matplotlib and make function calls directly to plot data, many Pandas methods for Series and DataFrame objects make this easier. The documentation gives a nice overview of this integration here (https://pandas.pydata.org/pandas-docs/stable/visualization.html) with more examples than provided here.

Let's begin by trying to make a pie chart from a Pandas Series.

In [1]:
```
import pandas as pd
from pandas import Series, DataFrame
```

In [2]:
```
# first we'll create a Series with three numbers
s = Series([5000000, 3000000, 2000000])
s
```

Out[2]:
```
0    5000000
1    3000000
2    2000000
dtype: int64
```

In [3]:
```
# there are a bunch of methods of the form Series.plot.METHOD for plotti
ng.
# suppose we want a pie plot:
s.plot.pie()
```

Out[3]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x11e0ca940>
```

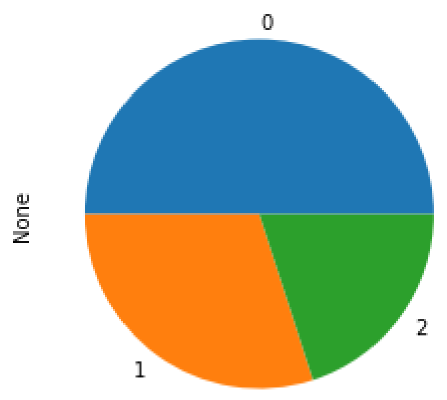
**Oops!** That's not what we wanted. We created a plot, but it didn't get rendered in the notebook. It turns out that matplotlib is integrated with Jupyter Notebooks, and sometimes we need a special command to tell Jupyter we want to render plots inline. Special Jupyter commands begin with a percent sign ("%"). We recommend putting the following at the beginning of all your notebooks (it's a Jupyter command, not Python code, so it won't work in a regular .py file if you were to try that):

In [4]:
```
%matplotlib inline
```

Ok, let's try plotting again.

```
In [5]: s.plot.pie()
```

```
Out[5]: <matplotlib.axes._subplots.AxesSubplot at 0x120204940>
```



Now we're getting somewhere! Of course, there are still many issues with this plot (you should adopt the mindset of a critic when we're making plots):

1. the font is tiny
2. we can only see the relative portions, not the absolute amounts
3. it says "None" to the left
4. each slice is numbered (not labeled)
5. there's no indicated of what is being measured

## A Better Plot

Let's address some of the issues we just saw. First, let's increase the font size. To do this, we'll import matplotlib directly, and change the default size. All the defaults are in a dictionary named `rcParams` in the `matplotlib` module.

```
In [6]: import matplotlib
        matplotlib.rcParams["font.size"]
```

```
Out[6]: 10.0
```

Let's increase to size 16.

```
In [7]: matplotlib.rcParams["font.size"] = 18
```

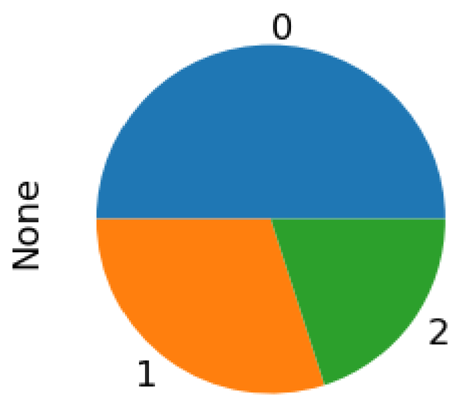
Second, we can pass a `figsize` tuple argument to specify the (width, height) in inches. Let's make the pie chart a 6-by-6 inch square.

```
In [8]: s = Series([5000000, 3000000, 2000000])
        s
```

```
Out[8]: 0    5000000
        1    3000000
        2    2000000
        dtype: int64
```

```
In [9]: s.plot.pie()
```

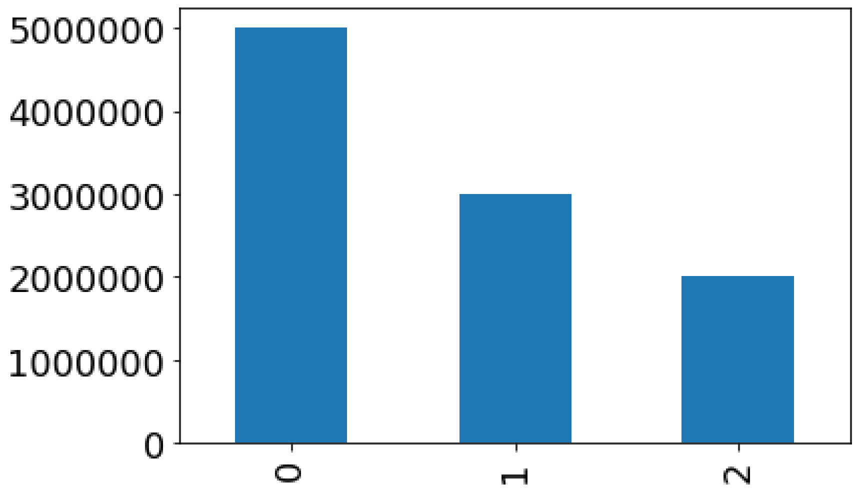
```
Out[9]: <matplotlib.axes._subplots.AxesSubplot at 0x120293c18>
```



Great! What about the absolute quantities? 95% of the time, it's best to replace a pie chart with a bar plot.

```
In [10]: s.plot.bar()
```

```
Out[10]: <matplotlib.axes._subplots.AxesSubplot at 0x1202e47f0>
```



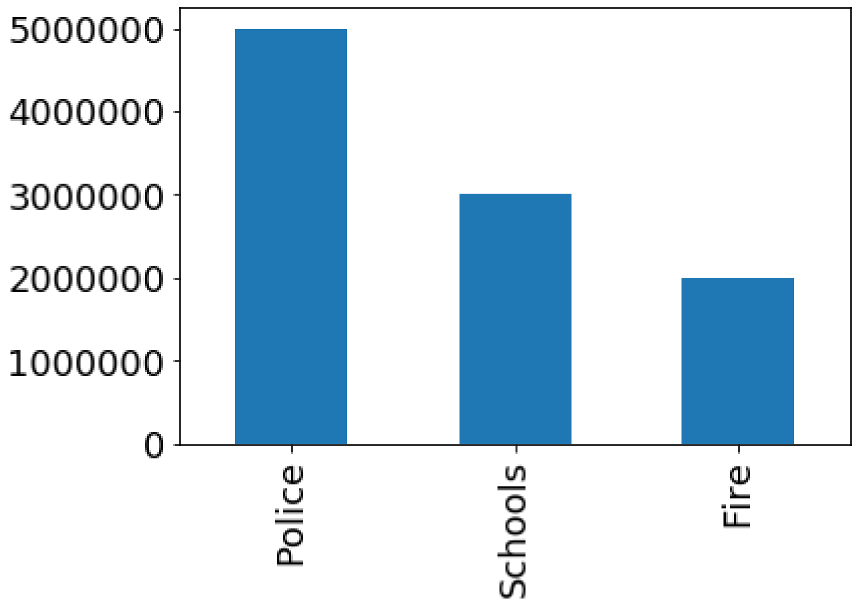
Those x-axis labels are coming from the Series index, which goes 0, 1, 2 (because we created the Series from a list). Let's create the Series from a dictionary to get better categories on the x-axis.

In [11]:
```
s = Series({"Police":5000000, "Schools":3000000, "Fire":2000000})
s
```

Out[11]:
```
Police     5000000
Schools    3000000
Fire       2000000
dtype: int64
```

In [12]:
```
s.plot.bar()
```

Out[12]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x12038d748>
```

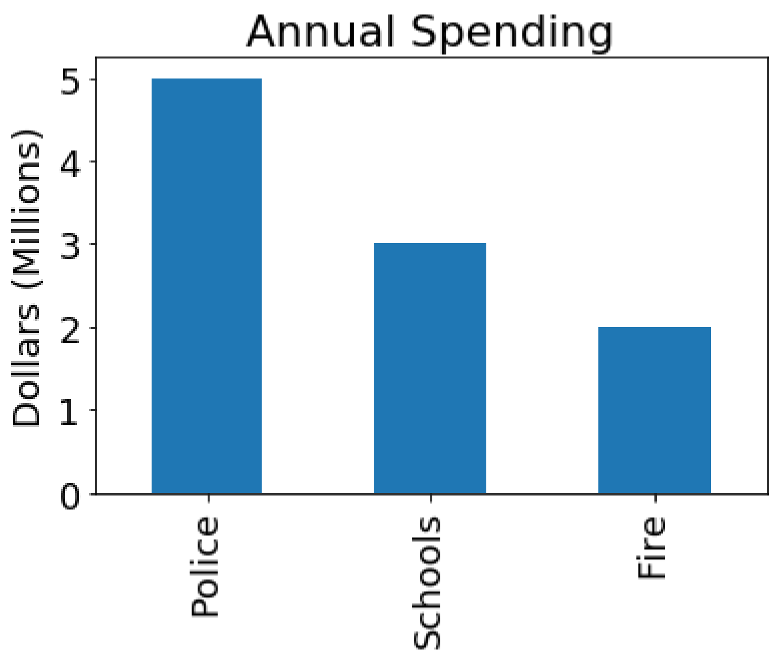


What is the y-axis measuring, and how big are those numbers? Counting zeros is annoying!

Whenever we call `Series.plot.plotting_function` (where `plotting_function` might be `pie` , `bar` , or similar), it returns an AxesSubplot object. We can call various methods on that to tweak the plot.

In [13]:
```
millions = s / 1e6
ax = millions.plot.bar()
ax.set_title("Annual Spending")
ax.set_ylabel("Dollars (Millions)")
```

Out[13]:
```
Text(0, 0.5, 'Dollars (Millions)')
```

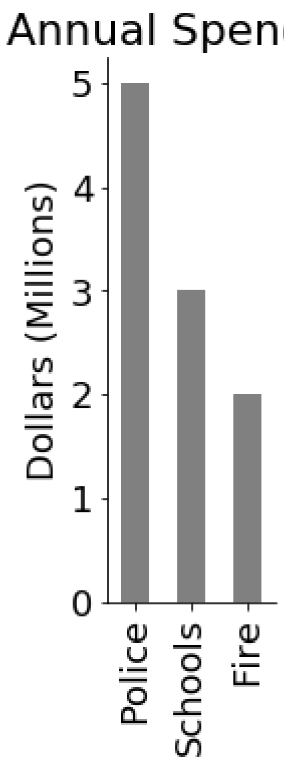


The above is a fine plot, but remember we're being critics! A few things would help:

1. it's best to be grayscale (more consistent when printed)
2. it's best to minimize non-data ink (why box everything in?)
3. the plot shouldn't be bigger than necessary

You should all read Edward Tufte's books (over break?) to start forming your philosophy of plotting: https://www.edwardtufte.com/tufte/books_vdqi (https://www.edwardtufte.com/tufte/books_vdqi)

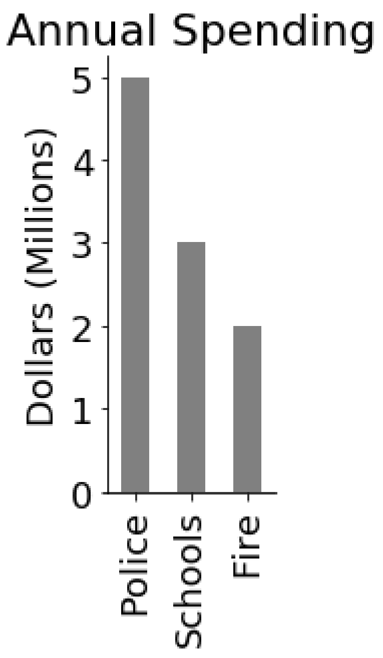
In [14]:
```
ax = millions.plot.bar(figsize=(1.5,5), color="0.5") # 0 is black, 1 is
 white, 0.5 is halfway between
ax.set_title("Annual Spending")
ax.set_ylabel("Dollars (Millions)")
ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)
```



Once you have a style you like, you should create a function so that many plots can be similar. One way to do this is have a function that creates an AxesSubplot object and returns it. The pandas plotting functions can this re-use this customized space. If we import pyplot from matplotlib, we can write such a function. For example:

In [15]:
```
from matplotlib import pyplot as plt

def get_ax(figsize=(4,4)):
    fig, ax = plt.subplots(figsize=figsize)
    ax.spines['right'].set_visible(False)
    ax.spines['top'].set_visible(False)
    return ax

ax=get_ax((1.5, 4))
ax.set_title("Annual Spending")
ax.set_ylabel("Dollars (Millions)")
millions.plot.bar(ax=ax, color="0.5")
```

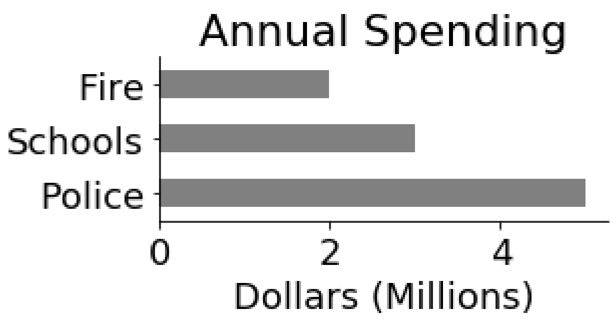
Out[15]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x12065f5c0>
```



We can also create a horizontal bar plot by replacing `bar` with `barh` and switching the axis-related calls:

In [16]:
```
ax=get_ax((4, 1.5))
ax.set_title("Annual Spending")
ax.set_xlabel("Dollars (Millions)")
millions.plot.barh(ax=ax, color="0.5")
```

Out[16]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x12069cb00>
```



## Example: Madison Route Distribution

In this example, we want to show how many people ride the 5 most popular bus routes in Madison, relative to overall ridership. We'll pull the data from our bus.db database we've used in previous examples.

In [17]:
```
import sqlite3
```

In [18]:
```
c = sqlite3.connect('bus.db')
```

In [19]:
```
# let's preview the data
pd.read_sql("SELECT * from boarding LIMIT 10", c)
```

Out[19]:

| | index | StopID | Route | Lat | Lon | DailyBoardings |
|---|---|---|---|---|---|---|
| **0** | 0 | 1163 | 27 | 43.073655 | -89.385427 | 1.03 |
| **1** | 1 | 1163 | 47 | 43.073655 | -89.385427 | 0.11 |
| **2** | 2 | 1163 | 75 | 43.073655 | -89.385427 | 0.34 |
| **3** | 3 | 1164 | 6 | 43.106465 | -89.340021 | 10.59 |
| **4** | 4 | 1167 | 3 | 43.077867 | -89.369993 | 3.11 |
| **5** | 5 | 1167 | 4 | 43.077867 | -89.369993 | 2.23 |
| **6** | 6 | 1167 | 10 | 43.077867 | -89.369993 | 0.11 |
| **7** | 7 | 1167 | 38 | 43.077867 | -89.369993 | 1.36 |
| **8** | 8 | 1169 | 3 | 43.089707 | -89.329817 | 18.90 |
| **9** | 9 | 1169 | 37 | 43.089707 | -89.329817 | 1.35 |

In [20]:
```
# we want to see the total ridership per bus route
df = pd.read_sql("SELECT Route, SUM(DailyBoardings) as ridership " +
                 "FROM boarding " +
                 "GROUP BY Route " +
                 "ORDER BY ridership DESC", c)

# let's peek at the first few rows in the results from our query
df.head()
```

Out[20]:

| | Route | ridership |
|---|---|---|
| **0** | 80 | 10211.79 |
| **1** | 2 | 4808.03 |
| **2** | 6 | 4537.02 |
| **3** | 10 | 4425.23 |
| **4** | 3 | 2708.55 |

Now's a good time to stop and think about what form the data is in, and what form we want to get it to.

**What we have:** a DataFrame of routes and ridership, indexed from 0.

**What we want:** a Series of the top 5 buses, with route numbers as the index, and ridership as the values.

Why do we want such a Series? Because when we call `Series.plot.bar(...)` we want a bar plot with five slices. Each pie should be labeled as a bus route (and slice labels are pulled from the index of a Series), and the size of the Series should correspond to ridership (and slice sizes are based on the values in a Series).

The first step to getting the data in the form we want is to re-index `df` so that the route numbers are in the index (instead of 0, 1, 2, etc). We can do this with the `DataFrame.set_index` function.

In [21]:
```
# set_index doesn't change df, but it returns a new
# DataFrame with the desired column as the new index
ridership_df = df.set_index("Route")
ridership_df.head()
```

Out[21]:

| | ridership |
|---|---|
| **Route** | |
| **80** | 10211.79 |
| **2** | 4808.03 |
| **6** | 4537.02 |
| **10** | 4425.23 |
| **3** | 2708.55 |

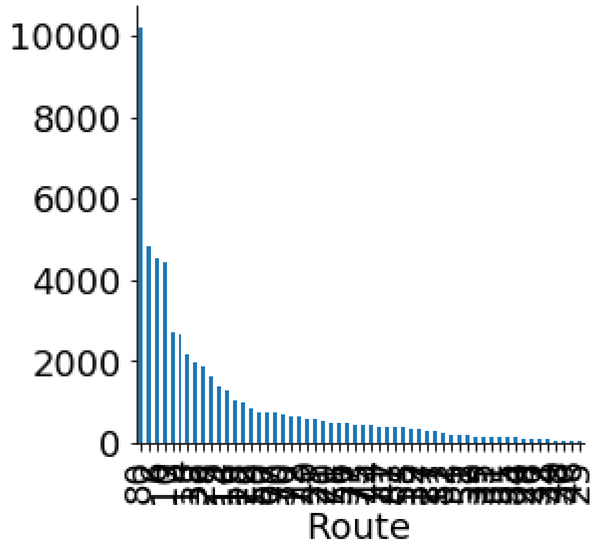
In [22]:
```
# we can pull the (only) ridership column from that DataFrame out
# and keep it as a Series.
ridership = ridership_df['ridership']
ridership.head()
```

Out[22]:
```
Route
80     10211.79
2       4808.03
6       4537.02
10      4425.23
3       2708.55
Name: ridership, dtype: float64
```

Great! Now we have the data in a plottable form. Let's make the pie chart.

```
In [23]: ridership.plot.bar(ax=get_ax())
```

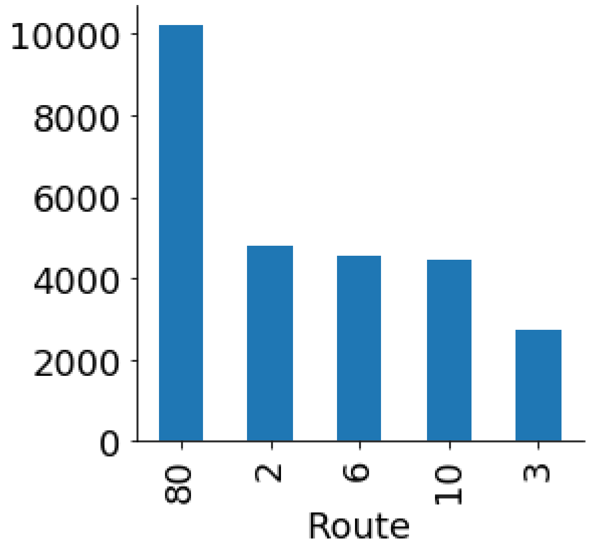
```
Out[23]: <matplotlib.axes._subplots.AxesSubplot at 0x1204a2b70>
```



This is somewhat close to the form we want. But we only wanted the top 5 routes (so that we can actually see what is going on!).

```
In [24]: ridership.head(5).plot.bar(ax=get_ax())
```

```
Out[24]: <matplotlib.axes._subplots.AxesSubplot at 0x12094b390>
```



Not bad, but we would ideally have an "other" category that captures all the routes besides the 80, 2, 6, 10, and 3. How many routes are in this other category?

```
In [25]: other_ridership = ridership[5:].sum()
         other_ridership
```

```
Out[25]: 29296.56
```

Now, we want to pull out the top 5 to a new Series, then add the other category.

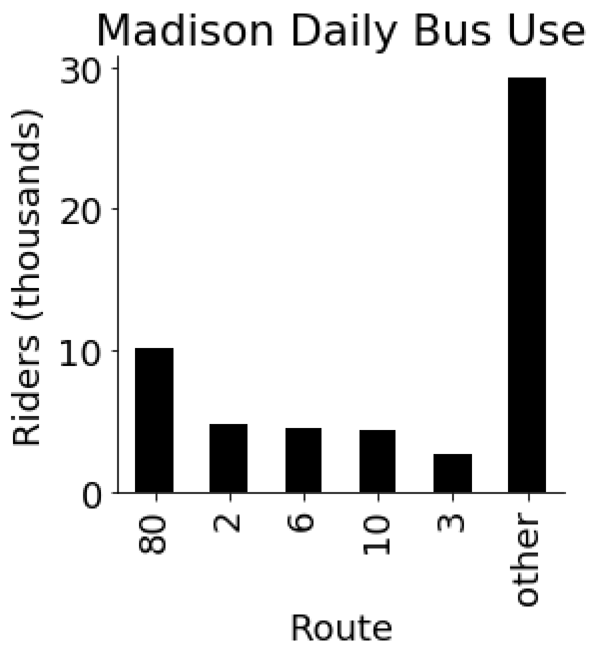
```
In [26]: top5 = ridership[:5]
         top5["other"] = other_ridership
         top5
```

```
Out[26]: Route
         80       10211.79
         2         4808.03
         6         4537.02
         10        4425.23
         3         2708.55
         other    29296.56
         Name: ridership, dtype: float64
```

That's exactly what we want! The ridership of the top 5 routes, and the remaining ridership spread across other routes. Let's plot it.

```
In [27]: ax = get_ax()
         (top5 / 1000).plot.bar(color="k", ax=ax) # "k" is black (because "b" was
         taken for blue)
         ax.set_ylabel("Riders (thousands)")
         ax.set_title("Madison Daily Bus Use")
```

```
Out[27]: Text(0.5, 1.0, 'Madison Daily Bus Use')
```



This is exactly what we want. We can see the top route (the 80) is responsible for about one fifth of the ridership. The top 5 routes together are responsible for almost half of all ridership (48%, to be exact). To wrap up, let's make sure we close our connection to bus.db.

```
In [28]: c.close()
```

## Scatter Plot

A scatter plot displays a collection of points along an x-axis and y-axis. Whereas pie charts are one-dimensional (we want to see a distribution of one value, such as ridership), scatter plots are naturally two dimensionals (each point has both an x and y position). Thus, scatter plots are generated from DataFrames (in contrast, pie charts are generated from a Series).

Just as there are a collection of `Series.plot.METHOD` methods, there are also a collection of `DataFrame.plot.METHOD` methods (`scatter` is one of those methods).

Let's begin by plotting some young trees. Each tree has an age (in years), a height (in feet), and a diameter (in inches).

In [29]:
```
trees = [
    {"age": 1, "height": 1.5, "diameter": 0.8},
    {"age": 1, "height": 1.9, "diameter": 1.2},
    {"age": 1, "height": 1.8, "diameter": 1.4},
    {"age": 2, "height": 1.8, "diameter": 0.9},
    {"age": 2, "height": 2.5, "diameter": 1.5},
    {"age": 2, "height": 3, "diameter": 1.8},
    {"age": 2, "height": 2.9, "diameter": 1.7},
    {"age": 3, "height": 3.2, "diameter": 2.1},
    {"age": 3, "height": 3, "diameter": 2},
    {"age": 3, "height": 2.4, "diameter": 2.2},
    {"age": 2, "height": 3.1, "diameter": 2.9},
    {"age": 4, "height": 2.5, "diameter": 3.1},
    {"age": 4, "height": 3.9, "diameter": 3.1},
    {"age": 4, "height": 4.9, "diameter": 2.8},
    {"age": 4, "height": 5.2, "diameter": 3.5},
    {"age": 4, "height": 4.8, "diameter": 4},
]
df = DataFrame(trees)
df
```

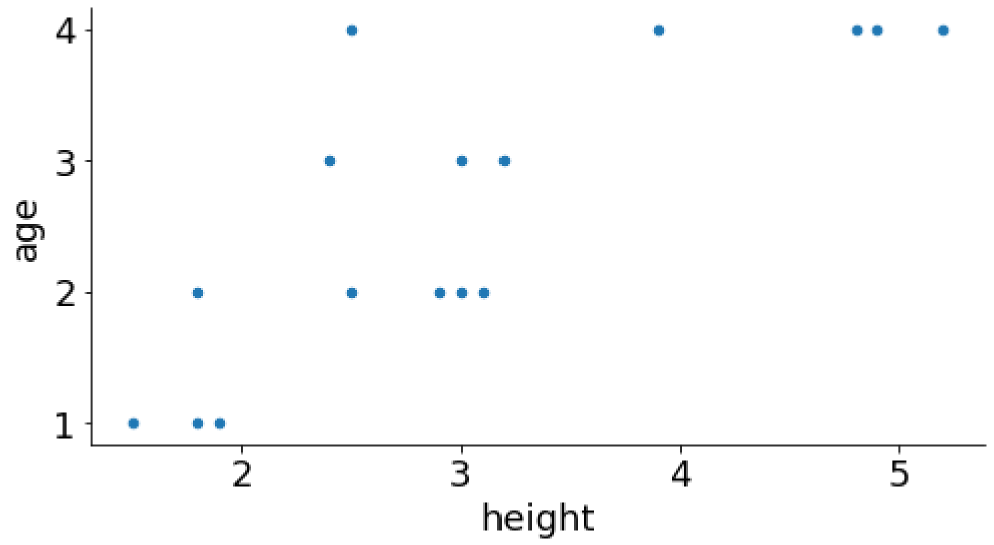
Out[29]:

| | age | diameter | height |
|---|---|---|---|
| **0** | 1 | 0.8 | 1.5 |
| **1** | 1 | 1.2 | 1.9 |
| **2** | 1 | 1.4 | 1.8 |
| **3** | 2 | 0.9 | 1.8 |
| **4** | 2 | 1.5 | 2.5 |
| **5** | 2 | 1.8 | 3.0 |
| **6** | 2 | 1.7 | 2.9 |
| **7** | 3 | 2.1 | 3.2 |
| **8** | 3 | 2.0 | 3.0 |
| **9** | 3 | 2.2 | 2.4 |
| **10** | 2 | 2.9 | 3.1 |
| **11** | 4 | 3.1 | 2.5 |
| **12** | 4 | 3.1 | 3.9 |
| **13** | 4 | 2.8 | 4.9 |
| **14** | 4 | 3.5 | 5.2 |
| **15** | 4 | 4.0 | 4.8 |

Let's plot this data and see if there seems to be any connection between tree age and tree height. We can create plots like this: `df.plot.scatter(x=FIELD1, y=FIELD2)`.

In [30]:
```
# you can choose which field is represented on the x-axis
# and which is represented on the y-axis.
df.plot.scatter(x='height', y='age', ax=get_ax(figsize=(8, 4)))
```

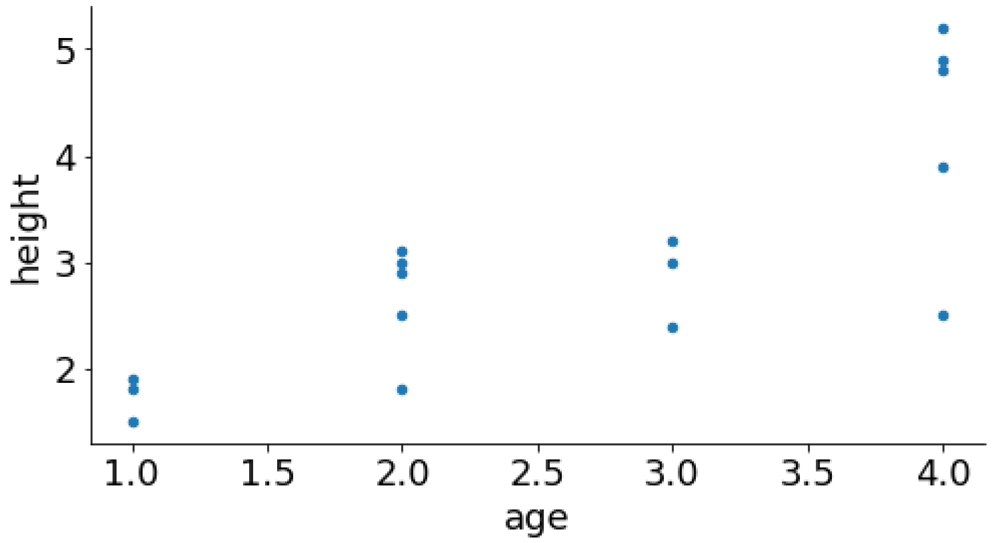
Out[30]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x120a622b0>
```



Although the above plot is informative (we can see that older trees are generally taller), it's not the easiest way to visualize the information. In general, people are accustomed to seeing time-related data on the x-axis (age is a type of time). Thus, a more intuitive plot would reverse the axes:

In [31]:
```
df.plot.scatter(x='age', y='height', ax=get_ax(figsize=(8, 4)))
```

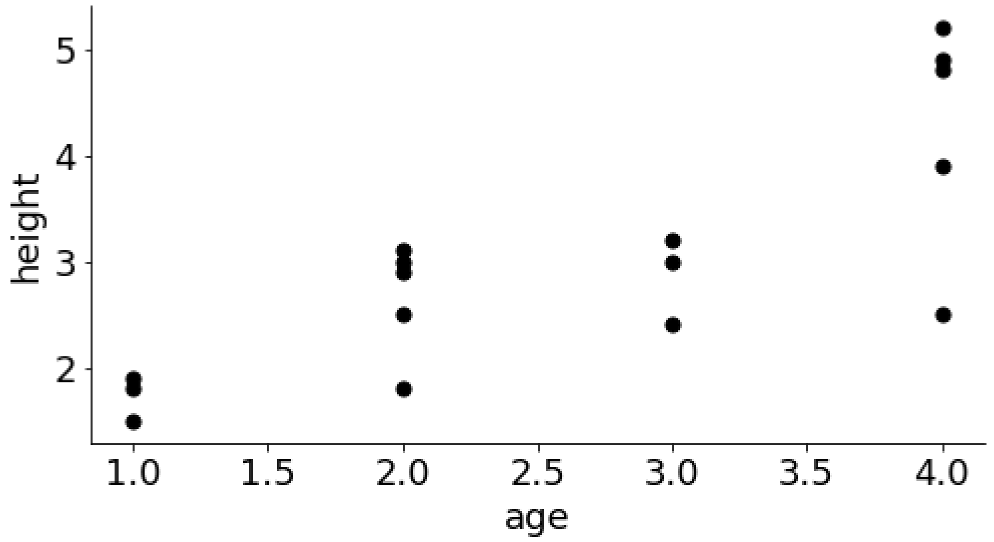
Out[31]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x120bac470>
```



We can also control the color (with the `c` argument) and size (with the `s` argument) of the points:

```
In [32]: df.plot.scatter(x='age', y='height', c='black', s=50, ax=get_ax(figsize=
         (8, 4)))
```

```
Out[32]: <matplotlib.axes._subplots.AxesSubplot at 0x120d27550>
```



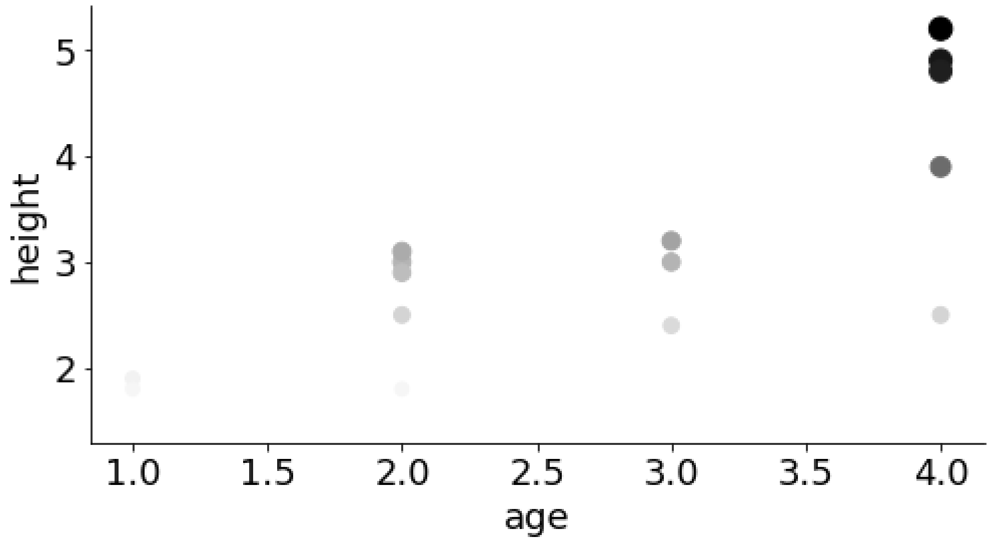
## Visually Communicating more Attributes

If we want, we can also use our data to determine the size and color of each point by passing a series for these. For example, suppose we want tall trees to be represented by large, black circles, and we wanted short trees to be represted with small gray dots. We can pull out a display Series to control this.

```
In [33]: display = df['height'] * 25
         display
```

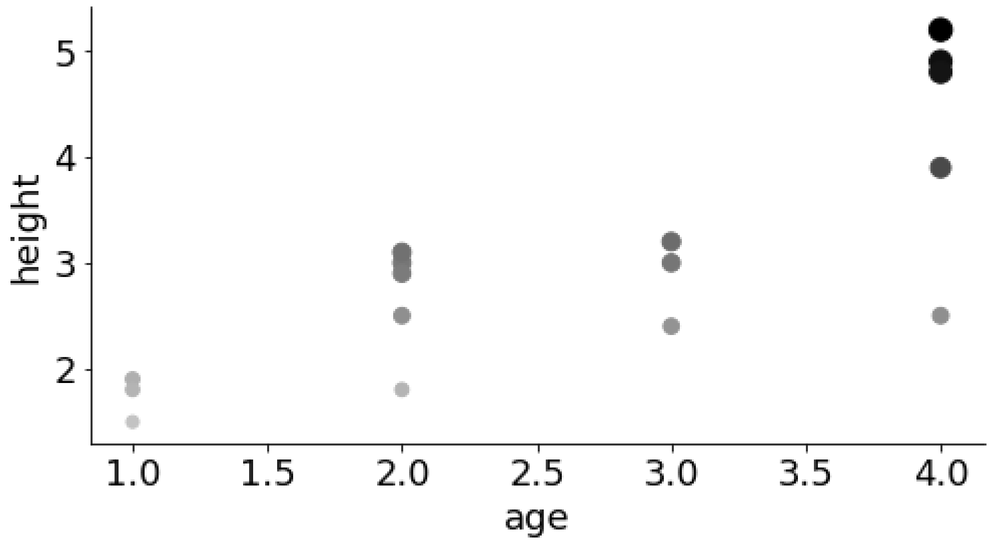
```
Out[33]: 0      37.5
         1      47.5
         2      45.0
         3      45.0
         4      62.5
         5      75.0
         6      72.5
         7      80.0
         8      75.0
         9      60.0
         10     77.5
         11     62.5
         12     97.5
         13    122.5
         14    130.0
         15    120.0
         Name: height, dtype: float64
```

```
In [34]: df.plot.scatter(x='age', y='height', c=display, s=display, ax=get_ax(fig
         size=(8, 4)))

Out[34]: <matplotlib.axes._subplots.AxesSubplot at 0x120b9c208>
```



matplotlib scaled the colors such that the smallest corresponds to white and the largest corresponds to black. This means we can't see the smallest on the white background. We can use `vmin` and `vmax` params to choose our own limits, making sure all data falls somewhere the visible range:

```
In [35]: ax = get_ax(figsize=(8, 4))
         df.plot.scatter(x='age', y='height', c=display, s=display, vmin=display.
         min()-50, ax=ax)

Out[35]: <matplotlib.axes._subplots.AxesSubplot at 0x1208ca7b8>
```



Bigger numbers in the `display` Series (or any Series used with `c=` ) results in darker dots, and bigger numbers also determine the size of the dots.

The above plot is an example of a **reduntant** visualization. It's reduntant because three characteristics of the plot (x-axis, dot color, and dot size) are all being used to communicate the same characteristic of the data (height). This is a wasteful use of plot characteristics, considering that we didn't communicate anything about tree diameter.

When you're thinking about how to visualize your data, you should list the interesting attributes of your data as well as the dimensions you can communicate with the plot. Then carefully consider how to use each dimension of your plot to communicate something interesting.

In this case, we have **three attributes** in our data:

1. tree age
2. tree height
3. tree diameter

We also have **four dimensions** we can control in our plot:

1. x-axis
2. y-axis
3. dot size
4. dot color

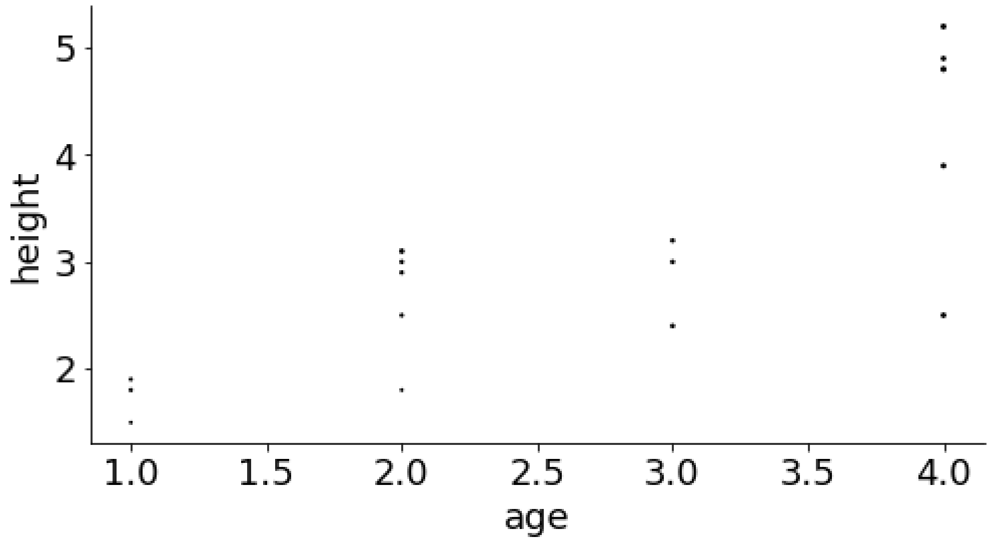
Which dimensions should communicate which attributes? Exact opinions will vary, but some combinations are more effective than others.

One good combination will be x-axis for age, y-axis for height, and dot size for diameter. As mentioned before, the x-axis is often used to communicate an elapsed time (an age). It also feels very natural for a vertical axis to communicate height. Finally, dot size seems like a better fit for diameter than color for two reasons. First, it is intuitive to pair a spatial attribute with a spatial characteristic of the plot. Second, color is often a tricky dimension to use. Gray can be too light to see, printed copies will very in how good they look, and finally you need to think about how accessible your plots will be for color-blind readers.

Let's see how the data looks with our final choices:

```
In [36]: ax = get_ax(figsize=(8, 4))
         df.plot.scatter(x='age', y='height', s=df['diameter'], c='black', ax=ax)
```

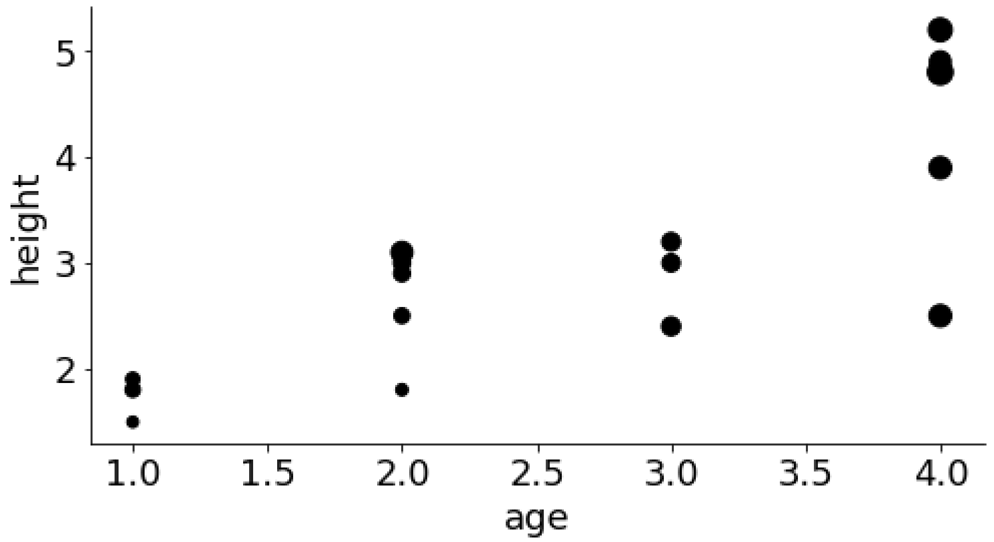
```
Out[36]: <matplotlib.axes._subplots.AxesSubplot at 0x120e4a780>
```



Those dots are a little small, and the visualization is relative (there's no scale specifying what a given dot size means), so we're free to multiply the diameter by a number of our choosing to make it more aesthetically pleasing.

```
In [37]: ax = get_ax(figsize=(8, 4))
         df.plot.scatter(x='age', y='height', s=df['diameter'] * 40, c='black', a
         x=ax)
```

```
Out[37]: <matplotlib.axes._subplots.AxesSubplot at 0x120fe82b0>
```



## Example: Iris Data

An Iris (https://en.wikipedia.org/wiki/Iris_%28plant%29) is a type of flowering plant. There is a very popular dataset (often used in machine learning examples) containing a description of the dimensions of 150 Iris plants from 3 different types of Iris.

The dataset is at https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data (https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data), and a description of the data is here (https://archive.ics.uci.edu/ml/datasets/iris). We will plot the data for each of the three types of Iris to visually identify patterns in the data.

As a first step, let's try fetching and loading the CSV data.

In [38]:
```
df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databa
ses/iris/iris.data')
df.head()
```

Out[38]:

| | 5.1 | 3.5 | 1.4 | 0.2 | Iris-setosa |
|---|---|---|---|---|---|
| **0** | 4.9 | 3.0 | 1.4 | 0.2 | Iris-setosa |
| **1** | 4.7 | 3.2 | 1.3 | 0.2 | Iris-setosa |
| **2** | 4.6 | 3.1 | 1.5 | 0.2 | Iris-setosa |
| **3** | 5.0 | 3.6 | 1.4 | 0.2 | Iris-setosa |
| **4** | 5.4 | 3.9 | 1.7 | 0.4 | Iris-setosa |

Oops! There's no CSV header in that file, so we have to tell Pandas what each field means. From the documentation, we see the following:

> 1. sepal length in cm
> 2. sepal width in cm
> 3. petal length in cm
> 4. petal width in cm
> 5. class:
>    -- Iris Setosa
>    -- Iris Versicolour
>    -- Iris Virginica

Let's try again using `header=`.

In [39]:
```
df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databa
ses/iris/iris.data',
                    names=['sepal-len', 'sepal-wid', 'petal-len', 'petal-wi
d', 'name'])
df.head()
```

Out[39]:

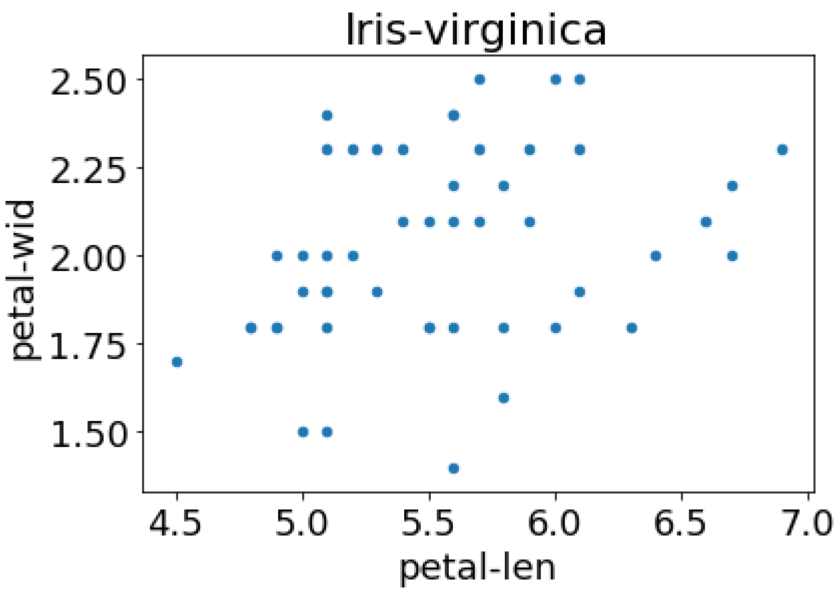
| | sepal-len | sepal-wid | petal-len | petal-wid | name |
|---|---|---|---|---|---|
| **0** | 5.1 | 3.5 | 1.4 | 0.2 | Iris-setosa |
| **1** | 4.9 | 3.0 | 1.4 | 0.2 | Iris-setosa |
| **2** | 4.7 | 3.2 | 1.3 | 0.2 | Iris-setosa |
| **3** | 4.6 | 3.1 | 1.5 | 0.2 | Iris-setosa |
| **4** | 5.0 | 3.6 | 1.4 | 0.2 | Iris-setosa |

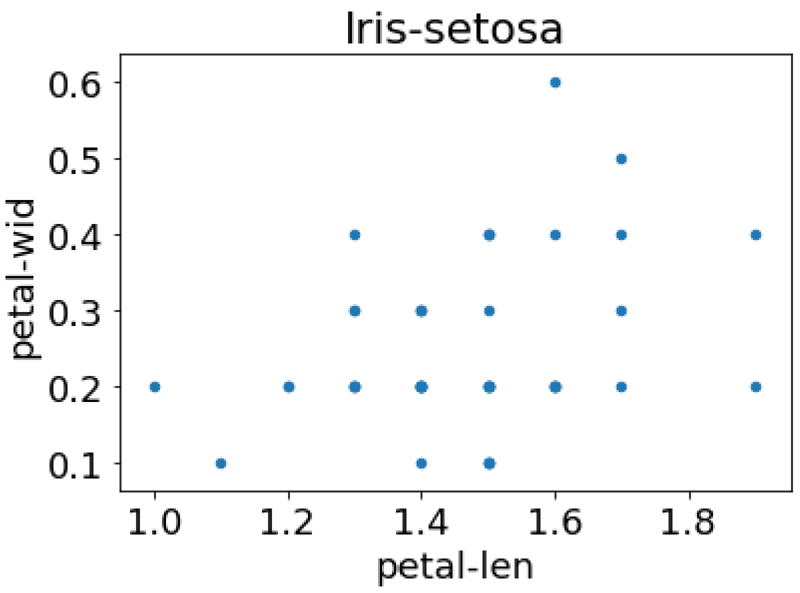
Great! Now lets see how many flowers there are of each type. We can use the `Series.value_counts` method on the `name` column Series. You should think of `value_counts` as equivalent to a `GROUP BY` with a `SUM` in SQL.

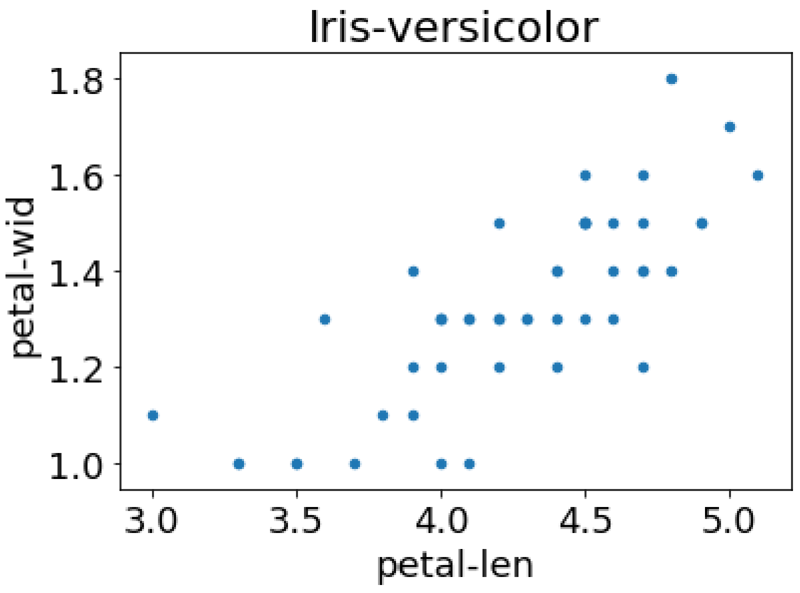
In [40]:
```
iris_types = df["name"].value_counts()
iris_types
```

Out[40]:
```
Iris-virginica     50
Iris-setosa        50
Iris-versicolor    50
Name: name, dtype: int64
```

In [41]:
```
for name in iris_types.index:
    rows = df[df['name'] == name]
    rows.plot.scatter(x='petal-len', y='petal-wid', title=name)
```


Iris-virginica
petal-wid
petal-len
2.50
2.25
2.00
1.75
1.50
4.5
5.0
5.5
6.0
6.5
7.0

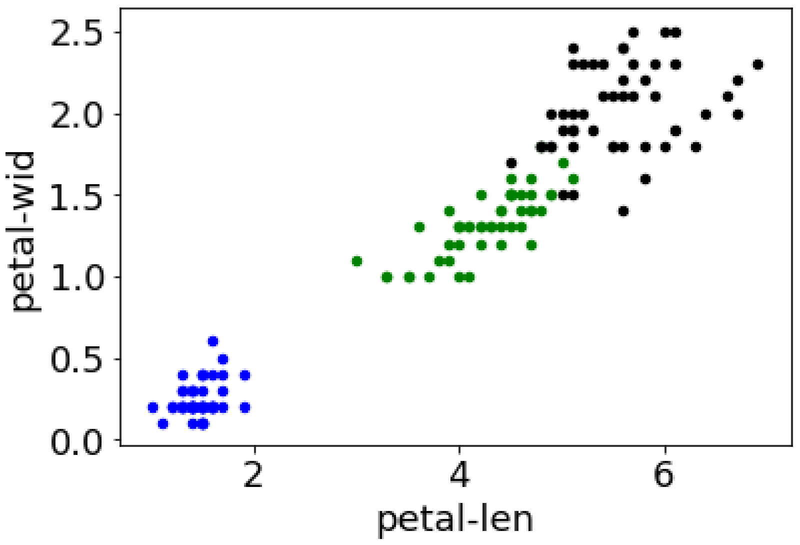

Iris-setosa
petal-wid
petal-len
0.6
0.5
0.4
0.3
0.2
0.1
1.0
1.2
1.4
1.6
1.8


Iris-versicolor
petal-wid
petal-len
1.8
1.6
1.4
1.2
1.0
3.0
3.5
4.0
4.5
5.0

*What if we want just one plot showing all three flower types?* In order to distinguish, we would want the dots for each flower type to be a different color. Not only can we pass the same `AxesSubplot` object to each plot call, but a call where we don't pass it returns a new one!

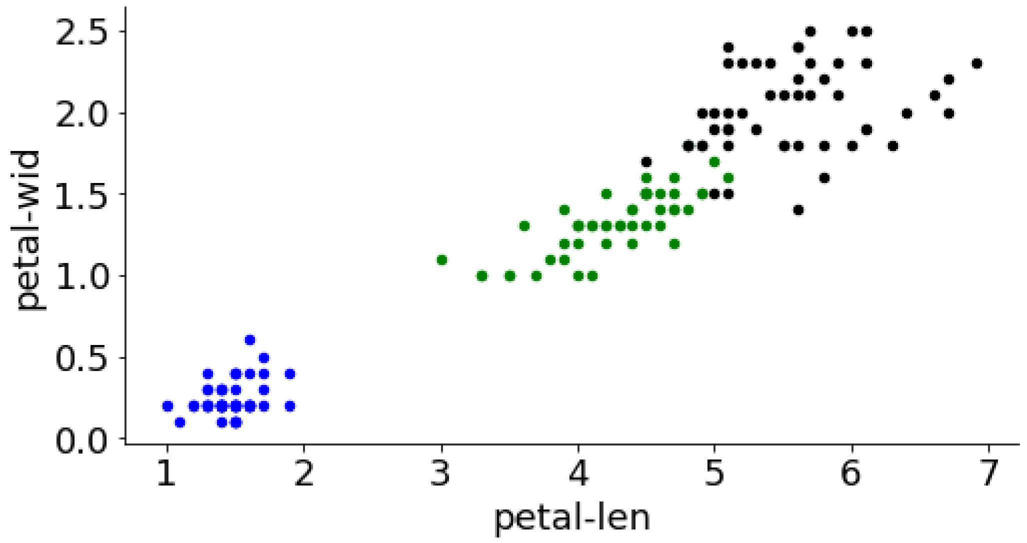
In [42]:
```
ax = df[df['name'] == 'Iris-setosa'].plot.scatter(x='petal-len', y='peta
l-wid', c='blue')
df[df['name'] == 'Iris-versicolor'].plot.scatter(x='petal-len', y='petal
-wid', c='green', ax=ax)
df[df['name'] == 'Iris-virginica'].plot.scatter(x='petal-len', y='petal-
wid', c='black', ax=ax)
```

Out[42]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1222f3a58>
```



In [43]:
```
ax = get_ax(figsize=(8, 4))
df[df['name'] == 'Iris-setosa'].plot.scatter(x='petal-len', y='petal-wid', c='blue', ax=ax)
df[df['name'] == 'Iris-versicolor'].plot.scatter(x='petal-len', y='petal-wid', c='green', ax=ax)
df[df['name'] == 'Iris-virginica'].plot.scatter(x='petal-len', y='petal-wid', c='black', ax=ax)
```

Out[43]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x12239a3c8>
```



From this plot, we can make several observations:

1. petal length is generally correlated with petal width
2. the Setosa variety is smallest along both dimensions
3. the Virginica variety is largest along both dimensions

What about sepal size? We leave that as an exercise for you!

## Conclusion

In this reading, we have learned how to create a bar from a Pandas Series and a scatter plot from a Pandas DataFrame. We have also discussed the decision making process for choosing which plot characteristics should represent which data attributes. Finally, we learned how about AxesSuplot objects and how to use them to plot multiple datasets in the same region. We did all of this in the context of two example datasets: the Madison Metro dataset and the popular Iris dataset.

In [ ]: