Contents

Part I

Modules

Chapter 1

`csv`

Python's `csv` module in the standard library parses the lines in a CSV file and allows us to quickly extract the values we're interested in. Here we'll present an example to read data from a csv file:

```
1 import csv
2
3
4 filename = 'my_csv_file.csv'
5 with open(filename) as f:
6     reader = csv.reader(f)
```

After the above code, reader contains an iterator that associated with the file content. Each iterated element in reader is a list containing the content of each line in the original file, with each element is element separated by coma. Pay attention that if you close the file ("disconnect" the file object with the file), this `reader` will not be able to access the content in file.

1.1 `reader()`

Documentation This is a member function. It accepts any object which supports the iterator protocol (by providing these methods: `__iter__()`, `__next__()`) and returns a string each time its `__next__()` method is called (file object and list object are both suitable).

It will return a reader object associated with the file. The reader object will iterate over lines in the given csv file. You need to use a variable to hold the returned reader object.

Data in csv file has following structure: Rows separated by newline character, and record separated by a coma in a single line. For example:

```
course name, time, credit hour, grade
COP 4530, fall 19, 3, A
COP 4531, fall 19, 3, A
CDA 3101, fall 19, 3, A
COP 4610, fall 19, 3, A
COT 4420, fall 19, 3, A
```

Reader object somehow read each row and transfer them into a list of strings, each item separated by the coma is an element in the list. Reader object is iterable over rows. It has struture like:

```
["course name", "time", "credit hour", "grade"]
["COP 4530", "fall 19", "3", "A"]
["COP 4531", "fall 19", "3", "A"]
["CDA 3101", "fall 19", "3", "A"]
["COP 4610", "fall 19", "3", "A"]
["COT 4420", "fall 19", "3", "A"]
```

Each row read from the csv file is returned as a list of strings. No automatic data type conversion is performed.

Reader object behaves like an iterator: it has `__iter__()` and `__next__()` defined. When calling `__iter__()`, it will return itself.

Chapter 2

`datetime`

Documentation

This module is used when you need object that representing date and time. Such object have many applications, for example:

- pass in as variable to be ploted in matplotlib

2.1 `datetime`

This is a class defined in datetime module (Documentation). An object of `datetime` type is a single object containing all the information from a `date` object and a `time` object. It assumes that there are exactly 3600 * 24 seconds in every day.

It can be printed directly by pass it into `print()`.

2.1.1 `strptime()`

This is a method inside `datetime` class, it can be used to convert a string of date information to python's datetime type. It accepts two arguments: `date_string` and `format`. `date_string` contains the information of your date, `format` contains rules to parse the `date_string`. It will return a `datetime` object corresponding to the `date_string`, parsed according to `format`. For

example:

```
import datetime



date = datetime.datetime.strptime('2014x7x1', '%Yx%mx%d')
print(date)
```

Output:

```
2014-07-01 00:00:00
```

The `%Yx` part is telling python to parse the part of the string before the first `x` as a **four-digit** year (if lower case y is provided, then it will be **two-digit** year: `%yx`). Similarly, `%m` is to parse month, and `%d` is to parse the day of the month. The original values of year, month and day should be valid (including leap years), otherwise, `ValueError` will rise.

The `strptime()` method can take a variety of arguments to determine how to interpret the date. You can find more here.

2.2 `strptime()`

Chapter 3

`matplotlib`

3.1 `pyplot`

3.1.1 `plot()` (plot line)

3.1.1.1 Pass a sequence of number into `pyplot.plot()`

We start with a simple example. Plot a list of numbers.

```
1 import matplotlib.pyplot as plt
2
3 squares = [1, 4, 9, 16, 25]
4 plt.plot(squares)
5 plt.show()
```

We first imported a class in `matplotlib`. The class name is `pyplot`, we give it an alias `plt`. Then we created a list of numbers, then call member function `plt.plot()` and pass the list into the function. After that, the figure is shown by calling member function `plt.show()`. This function opens `matplotlib`'s viewer and displays the plot.

When you give `pyplot.plot()` a sequence of numbers, it assumes the first data point corresponds to an x-coordinate value of 0. Thus, in the above example, we are actually plotting following data points:

`(0, 1), (1, 4), (2, 9), (3, 16), (4, 25)`

3.1.1.2 Pass two sequences of number into `pyplot.plot()`

We can pass two sequences of number into `plot()`, which will represent the x value and y value. For example:

```
x_value = [1, 2, 3, 4, 5]
squares = [1, 4, 9, 16, 25]
plt.plot(x_value, squares, linewidth=5)
```

Two sequences passed into the `pyplot.plot()` function should have same dimension, otherwise, a `ValueError` will be thrown.

You can plot multiple data sets on the same xy coordinates. Just provide the corresponding x and y sequences. For example:

```
x_value = [1, 2, 3, 4, 5]
cubes = [1, 8, 27, 64, 125]
squares = [1, 4, 9, 16, 25]
plt.plot(x_value, squares, x_value, cubes, linewidth=5)
```

You can also divide them into two calls of `pyplot.plot()`:

```
x_value = [1, 2, 3, 4, 5]
cubes = [1, 8, 27, 64, 125]
squares = [1, 4, 9, 16, 25]
plt.plot(x_value, cubes, linewidth=5)
plt.plot(x_value, squares, linewidth=5)
```

They will still be ploted into the same figure. You have to mannualy arrange different data set. The first and second data set will be evaluated as `x1` and `y1`, if only one data set is provided, it will be evaluated as `y`, and use default `[0, 1, 2, 3, ...]` as `x`.

3.1.1.3 line width

The line width of the curve can be set by:

```
plt.plot(squares, linewidth=5)
```

3.1.2 `scatter()` (plot scatter)

3.1.2.1 Single point

To plot single point, use `pyplot.scatter()`. It accepts at lease two arguments, corresponding to the x and y coordinates of the point. For example:

```
plt.scatter(float(input("Scatter x:")), float(input("Scatter y:")), s=200)
```

The argument `s` is used to set the size of the scatter.

3.1.2.2 Sequence of plot

We can plot a series of points using `scatter()`. Just pass lists of x- and y-values:

```
plt.scatter([1, 2, 3, 4, 5], [5, 6, 7, 1, 4], s=100)
plt.show()
```

You must provide x- and y-value at the same time (no default value of x will be used, unlike `plot()`), otherwise, `TypeError` will be thrown. Also, the dimension of x- and y-value lists must be the same, other `ValueError` will be thrown.

3.1.2.3 Set outline color of points

Default points have outline. You can set the edge color to the one you want. For example:

```
plt.scatter(x_values, y_values, s=20, edgecolors='red')  # change to red
plt.scatter(x_values, y_values, s=20, edgecolors='none')  # remove outline
```

The order of the style arguments are not affecting the result.

3.1.2.4 Set fill color of points

The fill color of points can be set by `c`:

```
plt.scatter(x_values, y_values, s=20, edgecolors='red', c='none')
```

3.1.3 `fill_between()`

This method takes one series of x-values and two series of y-values, and fills the space between the two y-value series over the series of x-values. For example:

```
from matplotlib import pyplot as plt


y_1 = [3, 4, 5, 3, 2, 3, 4]
y_2 = [5, 4, 2, 2, 6, 7, 8]
plt.plot(y_1, c='red')
plt.plot(y_2, c='black')
plt.scatter(list(range(len(y_1))), y_1, c='red')
plt.scatter(list(range(len(y_2))), y_2, c='black')

plt.fill_between(list(range(len(y_1))), y_1, y_2, facecolor='red', alpha=0.1)
plt.show()
```

The `alpha` argument controls a color's transparency. 0: transparent, 1 (default) is completely opaque.

3.1.4 Member Functions to Adjust Figure Appearance

In this section, several member functions are presented to adjust figure appearance.

3.1.4.1 `.title()`

Add title of the image. The size of the title can also be adjusted.

```
plt.title("Square Numbers", fontsize=24)
```

3.1.4.2 .xlabel(), .ylabel()

Set the x label and y label

```
plt.xlabel("Value", fontsize=14)
```

3.1.4.3 .tick_params() (adjust tick label of axis)

```
plt.tick_params(axis='both', labelsize=14)
```

3.1.4.4 set x, y axis range

```
plt.axis([x_0, x_1, y_0, y_1])
```

3.1.4.5 remove axis

```
plt.axes().get_xaxis().set_visible(False)
plt.axes().get_yaxis().set_visible(False)
```

3.1.4.6 set color

Documentation When you set the color of a object (by passing arguments to `c`), you can either pass a string to indicate its color (e.g. `edgecolor='red'`), or a tuple with three decimal values (correspond to red, green and blue). The range of the value is from 0 to 1. It is calculated by deviding RGB value with 255. So, a RGB color of R42, G160, B151 should be `(0.164, 0.627, 0.592)`. Example:

```
plt.scatter(x_values, y_values, s=20, edgecolors=(0.16, 0.63, 0.59), c='none')
```

3.1.4.7 use colormap

Colormap is a series of colors in a gradient that moves from a starting to ending color. Colormap allows you to emphasize a pattern in the data. For example, you might make low values a light color and high values a darker color.

Take the square scatter as an example:

```
import matplotlib.pyplot as plt
x_values = [x for x in range(1, 100)]
y_values = [x**2 for x in x_values]

plt.scatter(x_values, y_values, c=y_values, cmap=plt.cm.Greens,
↪  edgecolor='none', s=40)

plt.show()
```

We first set `y_values` to `c`, this tells python to use color maps to set the color of the point with regard to the `y_values`. You can set the specific colormap you want to use by `cmap=plt.cm.xxx`, where `xxx` is the name of color map you wish to use. The names and visual effect of different colormaps can be found here.

3.1.5 `.figure` (create a new `Figure` object)

Documentation

`figure()` will create a new object of figure. Check the documentation page for parameter list. The object will also be passed to new_figure_manager in the backends, which allows to hook custom `Figure` classes into the pyplot interface.

To set the size of the figure:

```
plt.figure(figsize=(a, b))
```

The `figsize` parameter takes a tuple, which is the dimensions of the plotting window in inches. Python assumes that the screen resolution is 80 pixels per inch. You can use another parameter `dpi` to tell python the resolution of your screen:

```
plt.figure(dpi=128, figsize=(10, 6))
```

You can use a variable to hold the `Figure` object returned by this method. Then you can call method defined inside `Figure` class to further adjust your figure. For example: `fig.autofmt_xdate()` can

draw the date labels in x axis diagonally to prevent them from overlapping (first used in *Python Crash Course*, project 2).

3.1.6 Utility Member Functions

3.1.6.1 `savefig()`

You can save the figure by calling `.savefig()` member function. Example:

```
1 plt.savefig('squares_plot.svg', bbox_inches='tight')
```

The first argument is a filename for the plot image, which will be saved to the same directory as your python file. The second argument trims extra whitespace from the plot.

Chapter 4

`numpy`

4.1 Array in Numpy

The key to `NumPy` is the array object defined inside it. It is an n-dimensional array of homogeneous types, with many operations being performed in compiled code for performance (higher efficiency). Compared with standard python sequences, array in `NumPy` has following feature:

1. Size is fixed. Modifying the size means creating a new array
2. Data type stored in each entry of the array must be the same
3. More efficienct mathematical operations

We have many different ways to create a `NumPy` array. In general, any numerical data that is stored in an array-like container can be converted to a `NumPy` array through use of array function (`numpy.array()`). You just provide the sequence container as input.

Other functions you can use:

4.1.1 `numpy.zeros()`

Accepts a tuple or list which indicates the shape of the zero matrix. It will return a matrix of that shape filled with zeros.

4.1.2 numpy.ones()

Same as numpy.zeros(), the filled value will be 1.

4.1.3 numpy.arange()

4.1.3.1 arange(num)

Accepts number. It will return a 1-D array, containing values from 0 to num - 1. By default, the value type is the same type as the input number, but you can change it to other data types (the basic data types defined in numpy). Also, the default step size is 1. For example:

```
import numpy as np
x = np.arange(5.5)
print(x)
```

result:

```
[0. 1. 2. 3. 4. 5.]
```

4.1.3.2 arange(num, dtype=)

Similar with arange(num), excepts that the type of the filled value is determined by the flag dtype. For example: numpy.arange(num, dtype=numpy.float32).

4.1.3.3 arange(start, end)

Accepts two numbers. It will return a 1-D array, containing values from start to end - 1. Default data type will be determined by the type of start and end. If one of them is integer, another is float, then the returned array will contain float values. Default step size is also one.

4.1.3.4 arange(start, end, stepsize)

Similar with arange(start, end), you can add a third parameter which determines the step size.

4.1.4 numpy.linspace()

This function creates arrays containing elements that are spaced equally between the specified beginning and end values.

4.1.4.1 numpy.linspace(start, end)

It accepts two numbers, and it will return a 1-D array that contains 50 numbers that are spaced equally over [start, end]. 50 is the default number of elements created between [start, end].

4.1.4.2 numpy.linspace(start, end, num)

It is similar with the above one, but the number of elements is given at the third parameter num.

4.1.5 numpy.random.random()

This function accepts a tuple (or list) that define the shape of a matrix. It will return a matrix filled with random numbers (in range [0, 1)) in that shape.

4.2 Member function and attribute of numpy's array

4.2.1 .min()

Return the minimum element.

4.2.2 .ravel()

This function will return a 1-D array that contains all element of the calling array. The original array is not modified.

4.2.3 .reshape()

Accepts tuple or direct number indicating the shape you want to change into. It will return a **COPY** of the matrix in changed shape. The original array is not affected. Example:

```
import numpy as np
x = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])
print(x)
print(x.reshape(3, 3))
```

result:

```
[1 2 3 4 5 6 7 8 9]
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

4.2.4 `.resize()`

Similar with `.reshape()`. Except this function will modify the calling array in place.

4.2.5 `.shape`

This is an attribute of the array class. It defines the shape of the array. You can assign a tuple to it to change the shape of the array. The number of elements in the array should be able to fill in the shape you changed, otherwise a `ValueError` will arise. Example:

```
import numpy as np
x = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])
print(x)
x.shape = (3, 3)
print(x)
```

result:

```
[1 2 3 4 5 6 7 8 9]
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

4.2.6 `.sum()`

Return the summation of all elements.

4.2.7 `.transpose()`

This function will return a transposed version of the calling array. The original array is not modified.

4.3 Array operations

Basic operations apply element-wise. The result is a new array with the resultant elements. Example:

```
>>> a = np.arange(5)
>>> b = np.arange(5)
>>> a + b
array([0, 2, 4, 6, 8])
>>> a - b
array([0, 0, 0, 0, 0])
>>> a > 3
array([False, False, False, False, True])
```

Operations like `*=` and `+=` will modify the original array.

4.4 Linear Algebra Operations

4.4.1 Matrix multiplication

You can't use `*` to perform a matrix multiplication since it is done element-wise. You have to call a function defined in `numpy` to do the matrix multiplication: `numpy.dot(a, b)`.

4.4.2 Matrix transpose

You can transpose an array by calling its member function: `a.transpose()`. It will return a transposed `a`, original array will not be modified.

4.4.3 Matrix inverse

There is a function (`inv()`) that is defined in `numpy.linalg.inv()` that accepts an array and it will return the inverse of the array. You have to make sure the array is a `n` by `n` matrix.

4.4.4 Creating unit matrix

Function `numpy.linalg.eye()` can do this. You have to pass in the parameter that determines its size.

4.4.5 Calculating trace of a matrix

Function `numpy.trace()` can do this. You have to pass in the array you want to know trace of. If the matrix is not square, you'll get a "partial" trace.

4.4.6 Solve linear matrix equation

Function `numpy.linalg.solve()` can do this. For example, we have the following equations:

$$x_1 + 2x_2 = 5$$

$$3x_1 + 4x_2 = 7$$

Use matrix notation:

```
ax = y
|1 2| |x1| |5|
|3 4| |x2| |7|
```

To solve for x, we call `numpy.linalg.solve(a, y)`. It will return the solution matrix. Code:

```
import numpy as np
a = np.array([[1, 2], [3, 4]])
y = np.array([[5.], [7.]])

print(np.linalg.solve(a, y))
```

result:

```
[[-3.]
 [ 4.]]
```

4.4.7 Get eigenvalues/eigenvectors of matrix

Function `numpy.linalg.eig()` can do this. You pass in a matrix, it will return its eigen value and eigen vector.

Chapter 5

pygal

Documentation

5.1 Chart Configuration

Pygal is customized at chart level with the help of the `Config` class. You can view the complete configuration attributes in here (name of each configuration you can use for your chart, and how you can use it).

5.2 Bar

`Bar` is a nested class inside `pygal`. Use this to create an object of a histogram. Each object of histogram has its own data set, as well as the styling information stored as attributes. They can be manipulated by the member function (methods) defined in `pygal.Bar`. To declare a `Bar` object:

```
1 hist = pygal.Bar()
```

(without any parameters)

5.2.1 attributes

5.2.1.1 title

`title` stores the title of the histogram objects. You can set the title of the histogram directly by assigning this attribute with a string. Example:

```
hist.title = "Your Histogram Title"
```

Other attributes work the same way. You can assign them directly with the value you wish.

5.2.1.2 x_labels

This attribute will add x label. You can assign it with a list of strings, they will be used to label the data in x direction. For example, if your have added a list: `[5, 4, 3, 1, 6]` to the `Bar` object, and want to plot histogram based on this. You can add x labels corresponding to each value, for example:

```
hist = Bar()
worked_hour = [5, 4, 3, 1, 6]
hist.add('work hour', worked_hour)
hist.x_labels = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
```

When you render the histogram, these labels will be under each bar.

5.2.1.3 y_labels

You can also set y labels, but it can override the default label that reflect the y value.

5.2.1.4 x_title

5.2.1.5 y_title

5.2.2 member functions

5.2.2.1 .add()

This function accepts two arguments: `add('title', value)`. You can pass in positionally. First is title and the second is value. The title is the name of this group of data, and value is your data set (can be a single number, which will only have one bar).

You can add multiple data set to one `Bar` object. Notice that python will not check if there is a naming conflict (duplicate names are permitted).

Pay attention that value added should be number (int or float), otherwise, a `TypeError` will rise. When you work with files, you may read the data in string format, be sure to convert them to numbers.

The `title` should be filled. If you don't want to add title, use an empty string: ''.

The bar created is interactive. You can customize the tooltip (which is shown when you hover your mouse above the bar) by passing in a list of dictionaries into the `add()` method, instead of a list of values. Each element in the list is a dictionary with two key-value pairs. Key `'value'` will be associated with the value, and key `'label'` will be associated with customized tooltip you want.

Each dictionary in list of dictionaries can be expanded to give the bar image more functions. For example, you can add clickable links to the bar. Just wrap the url link in the dictionary with key `'xlink'` (its value is set to the url link).

5.2.2.2 .render_to_file()

This function will render the data in the `Bar` object and save it to a file. The filename is passed in when calling the function. For example:

```
1 hist.render_to_file('my_plot.svg')
```

5.3 i18n

Pygal's country codes (two-letter codes) are stored in this module. i18n is short for *internationalization*. The dictionary COUNTRIES inside this module contains the two-letter country codes as keys and the country names as values.

The i18n module is in pygal_maps_world (which is a separate module from pygal). You can access COUNTRIES in following way:

```
1 from pygal_maps_world.i18n import COUNTRIES
```

5.4 World

Pygal includes a World class (which is a chart type, just like Bar is also a chart type). This chart can help to plot global data sets on a map. To use it, we declare an object of this class:

```
1 import pygal
2
3
4 wm = pygal.maps.world.World()
```

Pay attention where is the class located. We can also simplify the expression:

```
1 from pygal.maps.world import World
2
3
4 wm = World()
```

Current version of PyCharm (2019.1.2) will report error that world and World is not found, but you can still use it.

5.4.1 Attributes

You can modify these attributes of the World object.

5.4.1.1 title

The title of the image:

```
from pygal.maps.world import World


wm = World()
wm.title = 'World Map'
```

5.4.2 Functions

5.4.2.1 add()

This is used to set which country is highlighted on the final chart, as well as adding numeric data to those countries. Similar with `Bar.add()`.

It accepts two parameters, the first one is a string that contains the name of the group. The second one can be a list containing 2-digit country code (in string format) (these countries will be grouped together and have same color on the final image). For example:

```
from pygal.maps.world import World


wm = World()
wm.title = 'North America'

wm.add('North America', ['ca', 'mx', 'us'])
```

The rendered file will "light" the corresponding region that is indicated in the list you passed in.

If you want to pass numeric data to theose countries, you use a dictionary instead of a list in the `add()` function. The key should be the 2-digit country code, value should be numeric value that you want to pass to the country. By default, pygal uses these numbers to shade the countries from light (less in value) to dark (more in value). Example:

```
from pygal.maps.world import World


wm = World()
wm.title = 'Populations of Countries in North America'
wm.add('North America', {'ca': 341260000, 'us': 309349000, 'mx': 113423000})
wm.render_to_file('North America Population.svg')
```

5.4.2.2 render_to_file()

This method accepts one parameter, which is the filename of the file you want to render. It can also be relative path or path. For example:

```
wm.render_to_file('americas.svg')
```

5.5 style

Documentation

This sub-module contains predefined ways (classes) to style your chart. When creating a new pygal chart object (for example, `hist = Bar()`), you can pass the `style` parameter with different objects declared based on the classes in `style`. For example:

```
from pygal.style import LightSolarizedStyle
chart = pygal.StackedLine(fill=True, style=LightSolarizedStyle)
```

or:

```
from pygal.style import RotateStyle
from pygal.maps.world import World

world_map_style = RotateStyle('#336699')
world_map = World(style=world_map_style)
```

5.5.1 RotateStyle

Documentation

Examples to use:

```
from pygal.style import RotateStyle
from pygal.maps.world import World

world_map_style = RotateStyle('#336699')
world_map = World(style=world_map_style)
```

This `RotateStyle` class takes one argument, an RGB color in hex format. Pygal then chooses colors for each of the groups based on the color provided. The format of the RGB color is:

```
#rrggbb
rr: red component of the color
gg: green component of the color
bb: blue component of the color
The range of the color is 00 to FF
```

The style object can be used when creating a pygal chart object. You just need to pass it as a keyword argument when you declare a chart object (`style=style_object`).

Chapter 6

`random`

6.1 `choice()`

Documentation

`choice()` is a member function defined in `random` module. It has several overloads.

6.1.1 `choice(sequence)`

It accepts a sequence type, and will return a random element in the passed in sequence. If the sequence is empty, an `IndexError` will be thrown.

6.2 `randint()`

Documentation It accepts two arguments a, b. It will return a random integer `N`, such that `a <= N <= b`.

It is alias for `randrange(a, b + 1)`.

Chapter 7

`request`

Documentation

This module is used to work with API requests.

7.1 `get()`

Documentation

This function accepts an url, it will send a GET request and return the response object. You can hold the response object using a variable. This object contains a server's response to an HTTP request. For example:

```
# make an API call and store the response to a variable
url =
↪ 'https://api.github.com/search/repositories?q=language:python&sort=stars'
r = requests.get(url)
print("Status code:", r.status_code)
```

Output:

```
Status code: 200
```

The member `status_code` contains the status of accessing the url. 200 means normal access. (if you visit google from mainland China, you will get 404).

7.2 `Response`

This is a class that contains a server's response to an HTTP request.