MODULE-3

DICTIONARIES, NUMPY AND FILES

SYLLABUS:
Dictionaries: *Dictionary operations, dictionary methods, aliasing and copying.*
Numpy: *About, Shape, Slicing, masking, Broadcasting, dtype.*
Files: *About files, writing our first file, reading a file line-at-a-time, turning a file into a list of lines, Reading the whole file at once, working with binary files, Directories, fetching something from the Web.*

CHAPTERS: *5.4, 6.1-6.5, 7.1-7.8*

TEXT BOOK: *Peter Wentworth, Jeffrey Elkner, Allen B. Downey and Chris Meyers- How to think like a computer scientist: learning with python 3. Green Tea Press, Wellesley, Massachusetts, 2020*

DICTIONARIES

- All of the compound data types we have studied in detail so far — strings, lists, and tuples — are sequence types, which use integers as indices to access the values they contain within them.
- Dictionaries are yet another kind of compound type. They are Python's built-in mapping type. They map keys, which can be any immutable type, to values, which can be any type (heterogeneous), just like the elements of a list or tuple. In other languages, they are called associative arrays since they associate a key with a value.
- As an example, we will create a dictionary to translate English words into Spanish. For this dictionary, the keys are strings.
- One way to create a dictionary is to start with the empty dictionary and add key:value pairs. The empty dictionary is denoted {}:

```
>>> english_spanish = {}
>>> english_spanish["one"] = "uno"
>>> english_spanish["two"] = "dos"
```

- The first assignment creates a dictionary named english_spanish; the other assignments add new key:value pairs to the dictionary. We can print the current value of the dictionary in the usual way:

```
>>> print(english_spanish)
{"two": "dos", "one": "uno"}
```

- The key:value pairs of the dictionary are separated by commas. Each pair contains a key and a value separated by a colon.

HASHING

- The order of the pairs may not be what was expected. Python uses complex algorithms, designed for very fast access, to determine where the key:value pairs are stored in a dictionary. For our purposes we can think of this ordering as unpredictable.
- We might wonder why we use dictionaries at all when the same concept of mapping a key to a value could be implemented using a list of tuples:

```
>>> {"apples": 430, "bananas": 312, "oranges": 525, "pears": 217}
```

```
{'pears': 217, 'apples': 430, 'oranges': 525, 'bananas': 312}
>>> [('apples', 430), ('bananas', 312), ('oranges', 525), ('pears', 217)]
[('apples', 430), ('bananas', 312), ('oranges', 525), ('pears', 217)]
```

- The reason is dictionaries are very fast, implemented using a technique called hashing, which allows us to access a value very quickly. By contrast, the list of tuples implementation is slow. If we wanted to find a value associated with a key, we would have to iterate over every tuple, checking the 0th element. What if the key wasn't even in the list? We would have to get to the end of it to find out.
- Another way to create a dictionary is to provide a list of key:value pairs using the same syntax as the previous output:

```
>>> english_spanish = {"one": "uno", "two": "dos", "three": "tres"}
```

- It doesn't matter what order we write the pairs. The values in a dictionary are accessed with keys, not with indices, so there is no need to care about ordering. Here is how we use a key to look up the corresponding value:

```
>>> print(english_spanish["two"])
'dos'
```

The key "two" yields the value "dos".

- Lists, tuples, and strings have been called sequences, because their items occur in order. The dictionary is the first compound type that we've seen that is not a sequence, so we can't index or slice a dictionary.

DICTIONARY OPERATIONS

- The del statement removes a key:value pair from a dictionary. For example, the following dictionary contains the names of various fruits and the number of each fruit in stock:

```
>>> inventory = {"apples": 430, "bananas": 312, "oranges": 525, "pears": 217}
>>> print(inventory)
{'pears': 217, 'apples': 430, 'oranges': 525, 'bananas': 312}
```

- If someone buys all of the bananas, we can remove the entry from the dictionary:

```
>>> del inventory["bananas"]
>>> print(inventory)
{'apples': 430, 'oranges': 525, 'pears': 217}
```

- If we then try to see how many bananas we have, we get an error or if we're expecting more bananas soon, we might just change the value associated with bananas:

```
>>> inventory["bananas"] = 0
>>> print(inventory)
{'pears': 217, 'apples': 430, 'oranges': 525, 'bananas': 0}
```

- A new shipment of bananas arriving could be handled like this:

```
>>> inventory["bananas"] += 200
>>> print(inventory)
{'pears': 0, 'apples': 430, 'oranges': 525, 'bananas': 512}
```

- The len function also works on dictionaries; it returns the number of key:value pairs:

```
>>> len(inventory)
4
```

DICTIONARY METHODS

- Dictionaries have a number of useful built-in methods. The keys method returns what Python 3 calls a view of its underlying keys. A view object has some similarities to the range object we saw earlier, it is a lazy promise, to deliver its elements when they're needed by the rest of the program. We can iterate over the view, or turn the view into a list like this:

```
for key in english_spanish.keys(): # The order of the k's is not defined
    print("Got key", key, "which maps to value", english_spanish[key])
keys = list(english_spanish.keys())
print(keys)
```

This produces this output:

```
Got key three which maps to value tres
Got key two which maps to value dos
Got key one which maps to value uno
['three', 'two', 'one']
```

- It is so common to iterate over the keys in a dictionary that we can omit the keys method call in the for loop, iterating over a dictionary implicitly iterates over its keys:

```
for key in english_spanish:
    print("Got key", key)
```

- The values method is similar; it returns a view object which can be turned into a list:

```
>>> list(english_spanish.values())
['tres', 'dos', 'uno']
```

- The items method also returns a view, which promises a list of tuples— one tuple for each key:value pair:

```
>>> list(english_spanish.items())
[('three', 'tres'), ('two', 'dos'), ('one', 'uno')]
```

- Tuples are often useful for getting both the key and the value at the same time while we are looping:

```
for (key,value) in english_spanish.items():
    print("Got",key,"that maps to",value)
```

This produces:

```
Got three that maps to tres
Got two that maps to dos
Got one that maps to uno
```

- The in and not in operators can test if a key is in the dictionary:

```
>>> "one" in english_spanish
True
>>> "six" in english_spanish
False
>>> "tres" in english_spanish # Note that 'in' tests keys, not values.
False
```

- This method can be very useful, since looking up a non-existent key in a dictionary causes a runtime error:

```
>>> english_spanish["dog"]
Traceback (most recent call last):
    ...
KeyError: 'dog'
```

ALIASING AND COPYING

- As in the case of lists, because dictionaries are mutable, we need to be aware of aliasing. Whenever two variables refer to the same object, changes to one affect the other.
- If we want to modify a dictionary and keep a copy of the original, use the copy method. For example, opposites is a dictionary that contains pairs of opposites:

```
>>> opposites = {"up": "down", "right": "wrong", "yes": "no"}
>>> alias = opposites
>>> copy = opposites.copy() # Shallow copy
```

- alias and opposites refer to the same object; copy refers to a fresh copy of the same dictionary. If we modify alias, opposites is also changed:

```
>>> alias["right"] = "left"
>>> opposites["right"]
'left'
```

- If we modify copy, opposites is unchanged:

```
>>> copy["right"] = "privilege"
>>> opposites["right"]
'left'
```

COUNTING LETTERS

- Earlier we wrote a function that counted the number of occurrences of a letter in a string. A more general version of this problem is to form a frequency table of the letters in the string, that is, how many times each letter appears.
- Such a frequency table might be useful for compressing a text file. Because different letters appear with different frequencies, we can compress a file by using shorter codes for common letters and longer codes for letters that appear less frequently.
- Dictionaries provide an elegant way to generate a frequency table:

```
>>> letter_counts = {}
>>> for letter in "Mississippi":
```

```
...     letter_counts[letter] = letter_counts.get(letter, 0) + 1
...
>>> letter_counts
{'M': 1, 's': 4, 'p': 2, 'i': 4}
```

- We start with an empty dictionary. For each letter in the string, we find the current count (possibly zero) and increment it. At the end, the dictionary contains pairs of letters and their frequencies.
- It might be more appealing to display the frequency table in alphabetical order. We can do that with the items and sort methods (more precisely, sort orders lexicographically):

```
>>> letter_items = list(letter_counts.items())
>>> letter_items.sort()
>>> print(letter_items)
[('M', 1), ('i', 4), ('p', 2), ('s', 4)]
```

- Notice in the first line we had to call the type conversion function list. That turns the promise we get from items into a list, a step that is needed before we can use the list's sort method.

NUMPY

- The standard Python data types are not very suited for mathematical operations. For example, suppose we have the list a = [2, 3, 8]. If we multiply this list by an integer, we get:

```
>>> a = [2, 3, 8]
>>> 2 * a
[2, 3, 8, 2, 3, 8]
```

- And float's are not even allowed:

```
>>> a = [2, 3, 8]
>>> 2 * a
>>> 2.1 * a
TypeError: can't multiply sequence by non-int of type 'float'
```

- In order to solve this using Python lists, we would have to do something like:

```
values = [2, 3, 8]
result = []
for x in values:
    result.append(2.1 * x)
```

- This is not very elegant, is it? This is because Python list's are not designed as mathematical objects. Rather, they are purely a collection of items. In order to get a type of list which behaves like a mathematical array or matrix, we use Numpy.

```
>>> import numpy as np
>>> a = np.array([2, 3, 8])
>>> 2.1 * a
array([ 4.2, 6.3, 16.8])
```

- As we can see, this worked the way we expected it to. We note a couple of things: - We abbreviated numpy to np, this is conventional. - np.array takes a Python list as argument. - The list [2, 3, 8] contains int's, yet the result contains float's. This means numpy changed the data type automatically for us.
- Now let's take it a step further and see what happens when we multiply together array's.

```
>>> import numpy as np
>>> a = np.array([2, 3, 8])
>>> a * a
array([ 4, 9, 64])
>>> a**2
array([ 4, 9, 64])
```

This has nicely squared the array element-wise.

- Note: Those in the know might be a bit surprised by this. After all, if a is a vector, shouldn't a**2 be the dot product of the two vectors, $\vec{a} \cdot \vec{a}$? Well, numpy arrays are not vectors in the algebraic sense. Arithmetic operations between arrays are performed element-wise, not on the arrays as a whole.
- To tell numpy we want the dot product we simply use the np.dot function:

```
>>> a = np.array([2, 3, 8])
>>> np.dot(a,a)
77
```

- Furthermore, if you pass 2D arrays to np.dot it will behave like matrix multiplication. Several other similar NumPy algebraic functions are available (like np.cross, np.outer, etc.). When we want to treat numpy array operations as vector or matrix operations, make use of the specialized functions to this end.

SHAPE

- One of the most important properties an array is its shape. We have already seen 1 dimensional (1D) arrays, but arrays can have any dimensions you like. Images for example, consist of a 2D array of pixels. But in color images every pixel is an RGB tuple: the intensity in red, green and blue. Every pixel itself is therefore an array as well. This makes a color image 3D overall.
- To get the shape of an array, we use shape:

```
>>> import numpy as np
>>> a = np.array([2, 3, 8])
>>> a.shape
(3,)
```

- Something slightly more interesting:

```
>>> b = np.array([
    [2, 3, 8],
    [4, 5, 6],
    ])
>>> b.shape
    (2, 3)
```

SLICING

- Just like with lists, we might want to select certain values from an array. For 1D arrays it works just like for normal python lists:

```
>>> a = np.array([2, 3, 8])
>>> a[2]
8
>>> a[1:]
np.array([3, 8])
```

- However, when dealing with higher dimensional arrays something else happens:

```
>>> b = np.array([
    [2, 3, 8],
    [4, 5, 6],
    ])
>>> b[1]
array([4, 5, 6])
>>> b[1][2]
6
```

- We see that using b[1] returns the 1th row along the first dimenion, which is still an array. After that, we can select individual items from that. This can be abbreviated to:

```
>>> b[1, 2]
6
```

- But what if I wanted the 1th column instead of the first row? Then we use : to select all items along the first dimension, and then a 1:

```
>>> b[:, 1]
array([3, 5])
```

- By comparing with the definition of b, we see that this is the column we were looking for. Instead of first, I write 1th on purpose to signify the existence of a 0th element. Remember that in Python, as in any self-respecting programming language, we start counting at zero.

MASKING

- This is perhaps the single most powerful feature of Numpy. Suppose we have an array, and we want to throw away all values above a certain cutoff:

```
>>> a = np.array([230, 10, 284, 39, 76])
>>> cutoff = 200
>>> a > cutoff
np.array([True, False, True, False, False])
```

- Simply using the larger than operator lets us know in which cases the test was positive. Now we set all the values above 200 to zero:

```
>>> a = np.array([230, 10, 284, 39, 76])
>>> cutoff = 200
```

```
>>> a[a > cutoff] = 0
>>> a
np.array([0, 10, 0, 39, 76])
```

- The crucial line is a[a > cutoff] = 0. This selects all the points in the array where the test was positive and assigns 0 to that position. Without knowing this trick we would have had to loop over the array:

```
>>> a = np.array([230, 10, 284, 39, 76])
>>> cutoff = 200
>>> new_a = []
>>> for x in a:
>>>     if x > cutoff:
>>>         new_a.append(0)
>>>     else:
>>>         new_a.append(x)
>>> a = np.array(new_a)
```

BROADCASTING

- Another powerful feature of Numpy is broadcasting. Broadcasting takes place when you perform operations between arrays of different shapes. For instance:

```
>>> a = np.array([
    [0, 1],
    [2, 3],
    [4, 5],
    ])
>>> b = np.array([10, 100])
>>> a * b
array([[ 0, 100],
       [ 20, 300],
      [ 40, 500]])
```

- The shapes of a and b don't match. In order to proceed, Numpy will stretch b into a second dimension, as if it were stacked three times upon itself. The operation then takes place element-wise.
- One of the rules of broadcasting is that only dimensions of size 1 can be stretched (if an array only has one dimension, all other dimensions are considered for broadcasting purposes to have size 1). In the example above b is 1D, and has shape (2,). For broadcasting with a, which has two dimensions, Numpy adds another dimension of size 1 to b. b now has shape (1, 2). This new dimension can now be stretched three times so that b's shape matches a's shape of (3, 2).
- The other rule is that dimensions are compared from the last to the first. Any dimensions that do not match must be stretched to become equally sized. However, according to the previous rule, only dimensions of size 1 can stretch. This means that some shapes cannot broadcast and Numpy will give an error:

```
>>> c = np.array([
    [0, 1, 2],
    [3, 4, 5],
    ])
>>> b = np.array([10, 100])
```

```
>>> c * b
ValueError: operands could not be broadcast together with shapes (2,3) (2,)
```

- What happens here is that Numpy, again, adds a dimension to b, making it of shape (1, 2). The sizes of the last dimensions of b and c (2 and 3, respectively) are then compared and found to differ. Since none of these dimensions is of size 1 (therefore, unstretchable) Numpy gives up and produces an error.
- The solution to multiplying c and b above is to specifically tell Numpy that it must add that extra dimension as the second dimension of b. This is done by using None to index that second dimension. The shape of b then becomes (2,1), which is compatible for broadcasting with c:

```
>>> c = np.array([
    [0, 1, 2],
    [3, 4, 5],
    ])
>>> b = np.array([10, 100])
>>> c * b[:, None]
array([[ 0, 10, 20],
       [300, 400, 500]])
```

dtype

- A commonly used term in working with numpy is dtype - short for data type. This is typically int or float, followed by some number, e.g. int8. This means the value is integer with a size of 8 bits. As an example, let's discuss the properties of an int8.
- Each bit is either 0 or 1. With 8 of them, we have 28 = 256 possible values. Since we also have to count zero itself, the largest possible value is 255. The data type we have now described is called uint8, where the u stands for unsigned: only positive values are allowed. If we want to allow negative numbers we use int8. The range then shifts to -128 to +127.
- The same holds for bigger numbers. An int64 for example is a 64 bit unsigned integer with a range of -9223372036854775808 to 9223372036854775807. It is also the standard type on a 64 bits machine. You might think bigger is better. You'd be wrong. If you know the elements of your array are never going to be bigger than 100, why waste all the memory space? You might be better off setting your array to uint8 to conserve memory. In general however, the default setting is fine. Only when you run into memory related problems should you remember this comment.
- What happens when you set numbers bigger than the maximum value of your dtype?

```
>>> import numpy as np
>>> a = np.array([200], dtype='uint8')
>>> a + a
array([144], dtype=uint8)
```

- That doesn't seem right, does it? If you add two uint8, the result of 200 + 200 cannot be 400, because that doesn't fit in a uint8. In standard Python, Python does a lot of magic in the background to make sure the result is the 400 you would expect. But numpy doesn't, and will return 144. Why 144 is left as an exercise. To fix this, you should make sure that your numbers where not stored as uint8, but as something larger; uint16 for example. That way the resulting 400 will fit.

```
>>> import numpy as np
>>> a = np.array([200], dtype='uint16')
>>> a + a
```

```
array([400], dtype=uint16)
```

- By now must be thinking: so bigger is better after all! Just use the biggest possible int all the time, and you'll be fine! Apart from the fact that there is no biggest int, there is a bigger problem. If you work with images, each pixel from that image is stored as an RGB tuple: the intensity in red, green and blue. Each of these is a uint8 value for most standard formats such as .jpg and .png. For example, (0, 0, 0) will be black, and (255, 0, 0) is red. This means that when you load an image from your hard drive this dtype is selected for you, and if you are not aware of this, what will happen when you add an image to itself? (In other words, place two copies on top of each other) You might expect that everything will become more dense. Instead, you'll get noise because of the effect we just talked about.

ABOUT FILES

- While a program is running, its data is stored in random access memory (RAM). RAM is fast and inexpensive, but it is also volatile, which means that when the program ends, or the computer shuts down, data in RAM disappears. To make data available the next time the computer is turned on and the program is started, it has to be written to a non-volatile storage medium, such a hard drive, usb drive, or CD-RW.
- Data on non-volatile storage media is stored in named locations called files. By reading and writing files, programs can save information between program runs.
- Working with files is a lot like working with a notebook. To use a notebook, it has to be opened. When done, it has to be closed. While the notebook is open, it can either be read from or written to. In either case, the notebook holder knows where they are. They can read the whole notebook in its natural order or they can skip around.
- All of this applies to files as well. To open a file, we specify its name and indicate whether we want to read or write.

WRITING OUR FIRST FILE

- Let's begin with a simple program that writes three lines of text into a file:

```
with open("test.txt", "w") as myfile:
    myfile.write("My first file written from Python\n")
    myfile.write("---------------------------------\n")
    myfile.write("Hello, world!\n")
```

- Opening a file creates what we call a file handle. In this example, the variable myfile refers to the new handle object. Our program calls methods on the handle, and this makes changes to the actual file which is usually located on our disk.
- On line 1, the open function takes two arguments. The first is the name of the file, and the second is the mode. Mode "w" means that we are opening the file for writing.
- With mode "w", if there is no file named test.txt on the disk, it will be created. If there already is one, it will be replaced by the file we are writing.
- To put data in the file we invoke the write method on the handle, shown in lines 2, 3 and 4 above. In bigger programs, lines 2–4 will usually be replaced by a loop that writes many more lines into the file.
- The file is closed after line 4, at the end of the with block. A with block make sure that the file get close even if an error occurs (power outages excluded).

READING A FILE LINE-AT-A-TIME

- Now that the file exists on our disk, we can open it, this time for reading, and read all the lines in the file, one at a time. This time, the mode argument is "r" for reading:

```
with open("test.txt", "r") as my_new_handle:
    for the_line in my_new_handle:
        # Do something with the line we just read.
        # Here we just print it.
        print(the_line, end="")
```

- This is a handy pattern for our toolbox. In bigger programs, we'd squeeze more extensive logic into the body of the loop at line 5 — for example, if each line of the file contained the name and email address of one of our friends, perhaps we'd split the line into some pieces and call a function to send the friend a party invitation.
- On line 5 we suppress the newline character that print usually appends to our strings with end="". Why? This is because the string already has its own newline: the for statement in line 2 reads everything up to and including the newline character.
- If we try to open a file that doesn't exist, we get an error:

```
>>> mynewhandle = open("wharrah.txt", "r")
FileNotFoundError: [Errno 2] No such file or directory: "wharrah.txt"
```

TURNING A FILE INTO A LIST OF LINES

- It is often useful to fetch data from a disk file and turn it into a list of lines. Suppose we have a file containing our friends and their email addresses, one per line in the file. But we'd like the lines sorted into alphabetical order. A good plan is to read everything into a list of lines, then sort the list, and then write the sorted list back to another file:

```
with open("friends.txt", "r") as input_file:
    all_lines = input_file.readlines()
all_lines.sort()
with open("sortedfriends.txt", "w") as output_file:
    for line in all_lines:
        outut_file.write(line)
```

- The readlines method in line 2 reads all the lines and returns a list of the strings.

READING THE WHOLE FILE AT ONCE

- Another way of working with text files is to read the complete contents of the file into a string, and then to use our string-processing skills to work with the contents.
- We'd normally use this method of processing files if we were not interested in the line structure of the file. For example, we've seen the split method on strings which can break a string into words. So here is how we might count the number of words in a file:

```
with open("somefile.txt") as f:
    content = f.read()
words = content.split()
print("There are {0} words in the file.".format(len(words)))
```

- Notice here that we left out the "r" mode in line 1. By default, if we don't supply the mode, Python opens the file for reading.

Your file paths may need to be explicitly named: *In the above example, we're assuming that the file somefile.txt is in the same directory as your Python source code. If this is not the case, you may need to provide a full or a relative path to the file. On Windows, a full path could look like "C:\\temp\\somefile.txt", while on a Unix system the full path could be "/home/jimmy/somefile.txt".*

AN EXAMPLE

- Many useful line-processing programs will read a text file line-at-a-time and do some minor processing as they write the lines to an output file. They might number the lines in the output file, or insert extra blank lines after every 60 lines to make it convenient for printing on sheets of paper, or extract some specific columns only from each line in the source file, or only print lines that contain a specific substring. We call this kind of program a filter.
- Here is a filter that copies one file to another, omitting any lines that begin with #:

```
def filter(oldfile, newfile):
    with open(oldfile, "r") as infile, open(newfile, "w") as outfile:
        for line in infile:
            # Put any processing logic here
            if not line.startswith('#'):
                outfile.write(line)
```

- On line 2, we open two files: the file to read, and the file to write. From line 3, we read the input file line by line. We write the line in the output file only if the condition on line 5 is true.

DIRECTORIES

- Files on non-volatile storage media are organized by a set of rules known as a file system. File systems are made up of files and directories, which are containers for both files and other directories.
- When we create a new file by opening it and writing, the new file goes in the current directory (wherever we were when we ran the program). Similarly, when we open a file for reading, Python looks for it in the current directory.
- If we want to open a file somewhere else, we have to specify the path to the file, which is the name of the directory (or folder) where the file is located:

```
>>> wordsfile = open("/usr/share/dict/words", "r")
>>> wordlist = wordsfile.readlines()
>>> print(wordlist[:6])
['\n', 'A\n', "A's\n", 'AOL\n', "AOL's\n", 'Aachen\n']
```

- This (Unix) example opens a file named words that resides in a directory named dict, which resides in share, which resides in usr, which resides in the top-level directory of the system, called /. It then reads in each line into a list using readlines, and prints out the first 5 elements from that list.
- A Windows path might be "c:/temp/words.txt" or "c:\\temp\\words.txt". Because backslashes are used to escape things like newlines and tabs, we need to write two backslashes in a literal string to get one! So the length of these two strings is the same! We cannot use / or \ as part of a filename; they are reserved as a delimiter between directory and filenames.

- When working with files in directories it is a good idea to let Python deal with all the slashes and escaping them. The os.path module does this for different operating systems. Using os.path. join("directory", "filename") will automatically return "directory/filename" on Unix/Linux, and "directory\\filename" on Windows. This can not only be of great help when moving code from one system to another, or when sharing with colleagues. It also means that you do not have to take care of it yourself, and it might save you some issues with string handling.
- The file /usr/share/dict/words should exist on Unix-based systems, and contains a list of words in alphabetical order.

WHAT ABOUT FETCHING SOMETHING FROM THE WEB?

- The Python libraries are pretty messy in places. But here is a very simple example that copies the contents at some web URL to a local file.

```
import urllib.request
url = "http://xml.resource.org/public/rfc/txt/rfc793.txt"
destination_filename = "rfc793.txt"
urllib.request.urlretrieve(url, destination_filename)
```

- The urlretrieve function— just one call— could be used to download any kind of content from the Internet. We'll need to get a few things right before this works:
  - The resource we're trying to fetch must exist! Check this using a browser.
  - We'll need permission to write to the destination filename, and the file will be created in the "current directory" - i.e. the same folder that the Python program is saved in.
  - If we are behind a proxy server that requires authentication, (as some students are), this may require some more special handling to work around our proxy. Use a local resource for the purpose of this demonstration!
  - We need to make sure that if we use any data from the web, that we check if the contents are still as we expect them to be. A website may change, or it may disappear. It can also be taken over by a new owner who might change the contents completely. So before you use data from the internet, make your program check if the data is what you want it to be, before executing any code or showing it to any important users!

- Here is a slightly different example using the requests module. This module is not part of the standard library distributed with python, however it is easier to use and significantly more potent than the urllib module distributed with python. Read requests documentation on http://docs.python-requests.org to learn how to install and use the module.
- Here, rather than save the web resource to our local disk, we read it directly into a string, and we print that string:

```
import requests
url = "http://xml.resource.org/public/rfc/txt/rfc793.txt"
response = requests.get(url)
print(response.text)
```

- Opening the remote URL returns the response from the server. That response contains several types of information, and the requests module allows us to access them in various ways. On line 5, we get the downloaded document as a single string. We could also read it line by line as follows:

```
import requests
url = "http://xml.resource.org/public/rfc/txt/rfc793.txt"
response = requests.get(url)
for line in response:
    print(line)
```