## Module-3

**Numpy** *(Pg 133 – 138)*: About, Shape, Slicing, masking, Broadcasting, dtype.

**Files** *(Pg 139 – 143)*: 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.

**Recursive** *(Pg 163 – 167)*: Recursive data structures, Processing recursive number lists, Case study: Fibonacci numbers, Example with recursive directories and files.

Chapters: ***6.1-6.5, 7.1-7.8, 10.2 – 10.5***

---

## Chapter 6: Numpy (Pg 133 – 138)

***About, Shape, Slicing, masking, Broadcasting, dtype.***

---

### *About*

- 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)***

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

- 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***.
- 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.
- This has nicely squared the array element-wise.

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

---

### *Shape*

- One of the most important properties an array is its shape.
- 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,)
```

```
>>> 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 dimension, 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.

---

## 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)
```

- When working with images this becomes even more obvious, because there we might have to loop over three dimensions before we can use the if/else.

---

## *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. 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 you 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***.
- 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***.
- 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)
```

- 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)
```

---

## Chapter 7: Files (Pg 139 – 143)

***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.***

---

## 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.

---

## 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"
```

---

## Introduction to Binary Files

- A Binary File is a file that stores data in binary format (0s and 1s) instead of human-readable text. These files are used to store:
    - Images
    - Audio files
    - Video files
    - Executable programs
    - Serialized objects
    - Database files

### Text File vs Binary File

| Text File | Binary File |
|---|---|
| • Human readable | • Not human readable |
| • Stored as characters | • Stored as bytes |
| • Example: .txt, .csv | • Example: .jpg, .mp3, .dat |
| • Uses encoding | • No encoding required |

- Binary files are opened using **'b'** mode in Python.

***Syntax: file = open("filename", "mode")***

### Opening Binary Files

| Mode | Description |
|---|---|
| rb | Read binary file |
| wb | Write binary file |
| ab | Append binary file |

| | |
|---|---|
| rb+ | Read & Write |
| wb+ | Write & Read |
| ab+ | Append & Read |

### Writing to Binary Files

- To write binary data, we use **write()** method.
- **Example**

```
f = open("data.bin", "wb")
f.write(b"Welcome to Python Binary Files")
f.close()
```

*"b"* indicates binary string, Data stored as bytes.

### Reading Binary Files

- Binary files are read using **read()** method.
- **Example**

```
f = open("data.bin", "rb")
data = f.read()
print(data)
f.close()
```

- **Output**: b'Welcome to Python Binary Files'

### Using "with" Statement

- Using *"with"* automatically closes file.

```
with open("data.bin", "rb") as f:
    data = f.read()
    print(data)
```

- **Advantages**
  - No need to close file
  - Safer
  - Cleaner code

### Working with "pickle" Module

- Python provides *"pickle"* module to store objects in binary files.
- **Example**

```
import pickle
list1 = [10, 20, 30, 40]
with open("list.dat", "wb") as f:
    pickle.dump(list1, f)
```

### Reading Objects from Binary File

- **Example**

```
import pickle
with open("list.dat", "rb") as f:
    data = pickle.load(f)
    print(data)
```

- **Output:** *[10, 20, 30, 40]*

### Writing Multiple Objects

```
import pickle
student1 = {"name":"Ram","marks":85}
student2 = {"name":"Shyam","marks":90}
with open("student.dat","wb") as f:
    pickle.dump(student1,f)
    pickle.dump(student2,f)
```

### Reading Multiple Objects

```
import pickle
with open("student.dat","rb") as f:
    while True:
        try:
            data = pickle.load(f)
            print(data)
        except EOFError:
            break
```

### File Pointer in Binary Files

- Python supports file pointer functions:
  - ***tell()*** - Returns current position
  - ***seek()*** - Moves file pointer
- **Example**

```
f = open("data.bin","rb")
print(f.tell())
f.seek(5)
print(f.tell())
f.close()
```

### Working with Images (Binary Files)

```
with open("image.jpg","rb") as f1:
    data = f1.read()
with open("copy.jpg","wb") as f2:
    f2.write(data)
```

### Advantages of Binary Files

- Faster processing
- Compact storage
- Supports complex data
- Efficient for large data

### *Disadvantages of Binary Files*

- Not human readable
- Hard to edit manually
- Platform dependency (sometimes)

### *EOFError in Python*

- EOFError stands for End Of File Error.
- EOFError occurs when Python tries to read data from a file but reaches the end of the file and no more data is available.

### *EOFError usually occurs when:*

- Using **pickle.load()** in binary files
- Using **input()** and no input is provided
- Reading beyond available data

### *Example*

```
import pickle
with open("student.dat","rb") as f:
    while True:
        data = pickle.load(f)
        print(data)
```

When file ends, Python throws:

```
EOFError: Ran out of input
```

### Solution (Using try-except)

```
import pickle
with open("student.dat","rb") as f:
    while True:
        try:
            data = pickle.load(f)
            print(data)
        except EOFError:
            break
```

### Complete Example Program:

### Student Record using Binary File

```
import pickle
student = {
    "name": "Ravi",
    "age": 21,
    "marks": 88
}
with open("student.dat","wb") as f:
    pickle.dump(student,f)
with open("student.dat","rb") as f:
    data = pickle.load(f)
    print(data)
```

---

## *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.

### *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.
- You can also explore the ***os.path*** module for other handy features that will help you handling files.
- The file ***/usr/share/dict/words*** should exist on Unix-based systems and contains a list of words in ***alphabetical order***.

---

## *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".
- If we are behind a proxy server that requires authentication, 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.
- Here is a slightly different example using the requests module.

```
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 4, 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)
```

## Glossary

**delimiter** A sequence of one or more characters used to specify the boundary between separate parts of text.

**directory** A named collection of files, also called a folder. Directories can contain files and other directories, which are referred to as *subdirectories* of the directory that contains them.

**file** A named entity, usually stored on a hard drive, floppy disk, or CD-ROM, that contains a stream of characters.

**file system** A method for naming, accessing, and organizing files and the data they contain.

**handle** An object in our program that is connected to an underlying resource (e.g. a file). The file handle lets our program manipulate/read/write/close the actual file that is on our disk.

**mode** A distinct method of operation within a computer program. Files in Python can be opened in one of four modes: read (`"r"`), write (`"w"`), append (`"a"`), and read and write (`"+"`).

**non-volatile memory** Memory that can maintain its state without power. Hard drives, flash drives, and rewritable compact disks (CD-RW) are each examples of non-volatile memory.

**path** A sequence of directory names that specifies the exact location of a file.

**text file** A file that contains printable characters organized into lines separated by newline characters.

**volatile memory** Memory which requires an electrical current to maintain state. The *main memory* or RAM of a computer is volatile. Information stored in RAM is lost when the computer is turned off.

## Exercises

1. Write a program that reads a file and writes out a new file with the lines in reversed order (i.e. the first line in the old file becomes the last one in the new file.)
2. Write a program that reads a file and prints only those lines that contain the substring `snake`.
3. Write a program that reads a text file and produces an output file which is a copy of the file, except the first five columns of each line contain a four digit line number, followed by a space. Start numbering the first line in the output file at 1. Ensure that every line number is formatted to the same width in the output file. Use one of your Python programs as test data for this exercise: your output should be a printed and numbered listing of the Python program.
4. Write a program that undoes the numbering of the previous exercise: it should read a file with numbered lines and produce another file without line numbers.
5. Write a program that takes the dictionary used above, and returns some of the words using 1337sp34k

---

## Chapter 10: Recursive (Pg 163 – 167)

***Recursive data structures, Processing recursive number lists, Case study: Fibonacci numbers, Example with recursive directories and files.***

---

## *Recursive data structures*

- ***Most of the Python data types we have seen can be grouped inside lists and tuples in a variety of ways.***
- Lists and tuples can also be nested, providing many possibilities for organizing data.
- The ***organization of data*** for the purpose of making it easier to use is called a ***data structure***.

- A ***nested number list*** is a list whose elements are either:
  - Numbers
  - Nested number lists
- Notice that the term, nested number list is used in its own definition.
- ***Recursive definitions*** like this are quite common in mathematics and computer science.
- ***Recursive definitions provide a concise and powerful way to describe recursive data structures that are partially composed of smaller and simpler instances of themselves***.
- Now suppose our job is to write a function that will sum all of the values in a nested number list. Python has a built-in function which finds the sum of a sequence of numbers:

  ***>>> sum([1, 2, 8])***

  ***11***
- For our ***nested number list***, however, ***sum*** will not work:

```
>>> sum([1, 2, [11, 13], 8])
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'list'
>>>
```

- The problem is that the third element of this list, [11, 13], is itself a list, so it cannot just be added to 1, 2, and 8.

---

## *Processing recursive number lists*

- To sum all the numbers in our recursive nested number list we need to
  - traverse the list
  - visiting each of the elements within its nested structure
  - adding any numeric elements to our sum, and
  - recursively repeating the summing process with any elements which are themselves sub-lists.
- Thanks to recursion, the Python code needed to sum the values of a nested number list is surprisingly short:

```
def recursive_sum(nested_number_list):
    total = 0
    for element in nested_number_list:
        if type(element) is list:
            total += recursive_sum(element)
        else:
            total += element
    return total
```

- The body of ***recursive_sum*** consists mainly of a "***for***" loop that traverses ***nested_number_list***. If ***element*** is a numerical value (the ***else*** branch), it is simply added to ***total***. If ***element*** is a list, then ***recursive_sum*** is called again, with the element as an argument. The statement inside the function definition in which the function calls itself is known as the ***recursive call***.
- The example above has a ***base case*** which does not lead to a ***recursive call***: the case where the element is not a (sub-) list. Without a base case, you'll have ***infinite recursion***, and your program will not work.
- An ***alternative solution, completely recursive***, would be the following. Notice that this implementation ***does not contain a "for"*** loop!

```
def recursive_sum(nested_number_list):
    if len(nested_number_list) == 0:
        return 0
    head, *tail = nested_number_list
    if isinstance(head, list):  # If head is a list....
        return recursive_sum(head) + recursive_sum(tail)
    else:
        return head + recursive_sum(tail)
```

- Recursion is truly one of the most beautiful and elegant tools in computer science.
- A slightly more complicated problem is finding the largest value in our nested number list:

```
def recursive_max(nested_list):
    largest = None
    first_time = True
    for element in nested_list:
        if type(element) is list:
            value = recursive_max(element)
        else:
            value = element
        if first_time or value > largest:
            largest = value
            first_time = False
    return largest
```

- The added twist to this problem is finding a value for initializing largest. We can't just use nested_list[0], since that could be either a element or a list. To solve this problem(at every recursive call) we initialize a Boolean flag (at line 4). When we've found the value of interest (at line 9) we check to see whether this is the initializing (first) value for largest, or a value that could potentially change largest.
- Again, here we have a base case at line 8. If we don't supply a base case, Python stops after reaching a maximum recursion depth and returns a run time error. See how this happens, by running this little script which we will call ***infinite_recursion.py:***

```
def recursion_depth(number):
    print("{0}, ".format(number), end="")
    recursion_depth(number + 1)
recursion_depth(0)
```

- After watching the messages flash by, you will be presented with the end of a long trace back that ends with a message like the following:

***Runtime Error:*** *maximum recursion depth exceeded...*

## Case study: Fibonacci numbers

The famous **Fibonacci sequence** 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 134, … was devised by Fibonacci (1170-1250), who used this to model the breeding of (pairs) of rabbits. If, in generation 7 you had 21 pairs in total, of which 13 were adults, then next generation the adults will all have bred new children, and the previous children will have grown up to become adults. So in generation 8 you'll have 13+21=34, of which 21 are adults.

This *model* to explain rabbit breeding made the simplifying assumption that rabbits never died. Scientists often make (unrealistic) simplifying assumptions and restrictions to make some headway with the problem.

If we number the terms of the sequence from 0, we can describe each term recursively as the sum of the previous two terms:

```
fib(0) = 0
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2)  for n >= 2
```

This translates very directly into some Python:

```
def fib(n):
    if n <= 1:
        return n
    t = fib(n-1) + fib(n-2)
    return t
```

This is a particularly inefficient algorithm, and this could be solved far more efficient iteratively:

```
import time
t0 = time.clock()
n = 35
result = fib(n)
t1 = time.clock()
print("fib({0}) = {1}, ({2:.2f} secs)".format(n, result, t1-t0))
```

We get the correct result, but an exploding amount of work!

```
fib(35) = 9227465, (10.54 secs)
```

---

## Example with recursive directories and files.

- The following program lists the contents of a directory and all its subdirectories.

```
import os
def get_dirlist(path):
    dirlist = os.listdir(path)
    dirlist.sort()
    return dirlist
def print_files(path, prefix = ""):
    if prefix == "":
        print("Folder listing for", path)
        prefix = "| "
    dirlist = get_dirlist(path)
    for file in dirlist:
        print(prefix+file)
        fullname = os.path.join(path, file)
        if os.path.isdir(fullname):
            print_files(fullname, prefix + "| ")
```

- Calling the function ***print_files*** with some folder name will produce output similar to this:

```
Folder listing for c:\python31\Lib\site-packages\pygame
| __init__.py
| arraydemo.py
| blit_blends.py
| camera.py
| cursors.py
| data
| | alien1.png
| | alien2.png
...
```

## Glossary

**base case** A branch of the conditional statement in a recursive function that does not give rise to further recursive calls.

**infinite recursion** A function that calls itself recursively without ever reaching any base case. Eventually, infinite recursion causes a runtime error.

**recursion** The process of calling a function that is already executing.

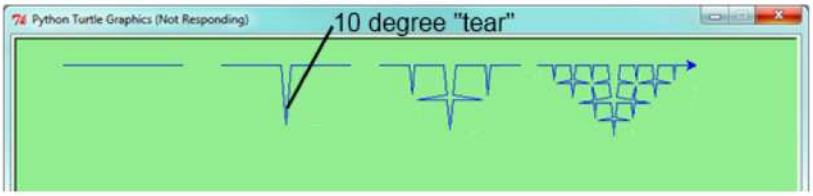
**recursive call** The statement that calls an already executing function. Recursion can also be indirect — function *f* can call *g* which calls *h*, and *h* could make a call back to *f*.

**recursive definition** A definition which defines something in terms of itself. To be useful it must include *base cases* which are not recursive. In this way it differs from a *circular definition*. Recursive definitions often provide an elegant way to express complex data structures, like a directory that can contain other directories, or a menu that can contain other menus.

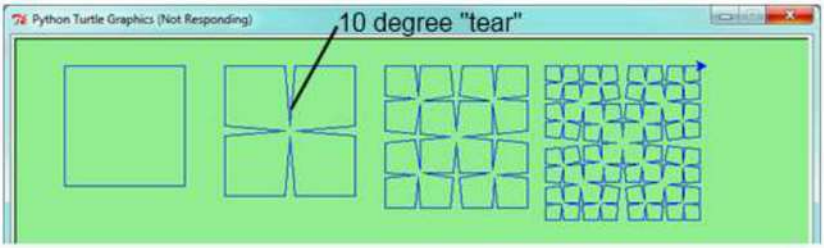
## Exercises

1. Modify the Koch fractal program so that it draws a Koch snowflake, like this:

2. a. Draw a Cesaro torn line fractal, of the order given by the user. We show four different lines of orders 0,1,2,3. In this example, the angle of the tear is 10 degrees.



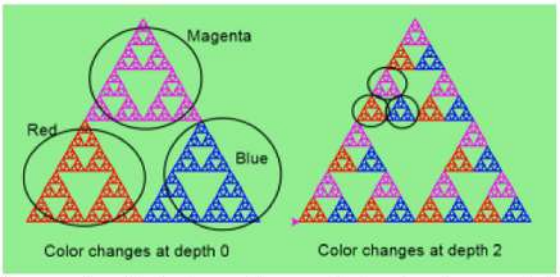

b. Four lines make a square. Use the code in part a) to draw cesaro squares. Varying the angle gives interesting effects — experiment a bit, or perhaps let the user input the angle of the tear.




c. (For the mathematically inclined). In the squares shown here, the higher-order drawings become a little larger. (Look at the bottom lines of each square - they're not aligned.) This is because we just halved the drawn part of the line for each recursive subproblem. So we've "grown" the overall square by the width of the tear(s). Can you solve the geometry problem so that the total size of the subproblem case (including the tear) remains exactly the same size as the original?

3. A Sierpinski triangle of order 0 is an equilateral triangle. An order 1 triangle can be drawn by drawing 3 smaller triangles (shown slightly disconnected here, just to help our understanding). Higher order 2 and 3 triangles are also shown. Draw Sierpinski triangles of any order input by the user.

4. Adapt the above program to change the color of its three sub-triangles at some depth of recursion. The illustration below shows two cases: on the left, the color is changed at depth 0 (the outmost level of recursion), on the right, at depth 2. If the user supplies a negative depth, the color never changes. (Hint: add a new optional parameter `colorChangeDepth` (which defaults to -1), and make this one smaller on each recursive subcall. Then, in the section of code before you recurse, test whether the parameter is zero, and change color.)




5. Write a function, `recursive_min`, that returns the smallest value in a nested number list. Assume there are no empty lists or sublists:

6. Write a function `count` that returns the number of occurrences of `target` in a nested list:

7. Write a function `flatten` that returns a simple list containing all the values in a nested list:

8. Rewrite the fibonacci algorithm without using recursion. Can you find bigger terms of the sequence? Can you find `fib(200)`?

9. Use help to find out what `sys.getrecursionlimit()` and `sys.setrecursionlimit(n)` do. Create several experiments similar to what was done in *infinite_recursion.py* to test your understanding of how these module functions work.

10. Write a program that walks a directory structure (as in the last section of this chapter), but instead of printing filenames, it returns a list of all the full paths of files in the directory or the subdirectories. (Don't include directories in this list — just files.) For example, the output list might have elements like this:

```
["C:\Python31\Lib\site-packages\pygame\docs\ref\mask.html",
 "C:\Python31\Lib\site-packages\pygame\docs\ref\midi.html",
 ...
 "C:\Python31\Lib\site-packages\pygame\examples\aliens.py",
 ...
 "C:\Python31\Lib\site-packages\pygame\examples\data\boom.wav",
 ... ]
```

11. Write a program named `litter.py` that creates an empty file named `trash.txt` in each subdirectory of a directory tree given the root of the tree as an argument (or the current directory as a default). Now write a program named `cleanup.py` that removes all these files.

    *Hint #1:* Use the program from the example in the last section of this chapter as a basis for these two recursive programs. Because you're going to destroy files on your disks, you better get this right, or you risk losing files you care about. So excellent advice is that initially you should fake the deletion of the files — just print the full path names of each file that you intend to delete. Once you're happy that your logic is correct, and you can see that you're not deleting the wrong things, you can replace the print statement with the real thing.

    *Hint #2:* Look in the `os` module for a function that removes files.

**---------- END OF MODULE 3 ----------**