Assignment 1
Introduction to Python

Applied Programming Lab (EE2703)

January 24, 2019

1 Introduction

In the first three semesters, you learned about C programming, some VERILOG and some assembly language programming. In this lab, you are going to learn to code in Python. Python is one of the most popular programming languages of today. It's simplicity in syntax, availability of a large number of open source libraries and the amazing open source community support makes it the favorite for scientists.

In this lab, we will learn how to use **Python 3** for scientific computing.

1.1 Introduction to Python

Let's start by trying to appreciate why python is so widely used. Let's look at a small example code to compute the Fibonacci series. Let's revise the algorithm first:

```
n=1
nold=1
print 1, nold
print 2, n
repeat for k=3,4,...,10
    new=n+nold
    nold=n
    n=new
    print k,new
end loop
```

Here is the code for the same in C:

```
1 #include<stdio.h>
```

```c
#define kmax 10
int main(void){
    int n,nold,new,k;
    n=1;nold=1;
    printf("%2d: %d\n",1,1);
    printf("%2d: %d\n",2,1);
    for( k=3;k<=10;k++ ){
        new=n+nold;
        nold=n;
        n=new;
        printf("%2d: %d\n",k,new);
    }
}
```

Let's see how the same code will look in Python:

```python
print("1 1")
print("2 1")
n=1;nold=1
for k in range(3,11):
    new=n+nold
    nold=n
    n=new
    print("%d %d" % (k, new))
```

Most of it is trivial to read and understand!

2 The Python Language

When it comes to programming languages, there has been a long history of both compiler based languages and interpreter based languages. On the compiler side, there was Fortran followed by Cobol and after quite a while, there was Pascal and then finally C. With each new language new features were added. Of course the older languages did not stay put and so there is a Fortran2003 which has many of the features of C++ that are relevant to scientific programming.

Similarly, on the interpreter side, there were early attempts which included Flex and Basic, as well as the language of the command shell. When UNIX came on the scene, the shell came with a sophisticated programming language. But UNIX was special in that it added special power to this shell via other utilities such as awk, sed, yacc etc. These were utilities of power never before seen by casual users, and many inventive uses were found for them. Around the same time, on the IBM side, an interesting language called REXX also appeared which first invented one of the most important modern data structures, namely associative arrays.

With so many utilities and options things were a jumble and soon users began to demand a more integrated language that had all these features built in. This ushered in the modern set

of languages of which the premier one is PERL. PERL introduced many things into interpreted languages, so many that it is hard to enumerate them. But it tried to do too much and it is a sprawling language, one in which an expert can do magic, but one which requires quite a lot of studying to master.

Following PERL were some competitors of whom I consider Ruby and Python the best. Both are lovely languages from the user's point of view and both are very similar in their features. Ruby has developed into a terrific language for database programming. Python has become the premier language for science and engineering. This is why we will focus on Python in this course. A third direction in which languages developed was the scratch pad type of language. It started in the sixties with early versions such as the Online Math System of UCLA. In the seventies, a powerful scientific development package called Basis was introduced. In the early eighties, IBM had introduced scientific spreadsheets for laboratory use. This branch has led to modern tools such as Matlab, Mathematica, and Labview.

Recently, a surprising development has taken place. People have developed modules (extra functionality) for Python that enables it to do most of the things that Matlab, Labview and Mathematica can do. This has created a special status for Python as far as technical users are concerned.

In this course, we will study Python. Along the way, most of the capabilities of Matlab will be demonstrated in Python, so that you can become a competent user of Matlab as well very quickly. In this assignment, the focus is on Python basics.

2.1 Hello World

Without breaking our tradition, let's start with a hello world program. Open a file and write `print("Hello World")` and save it as `hello.py`.

```
1 print("Hello World")
```

Listing 1: "hello.py"

Now, open the terminal and issue the command `python3 hello.py`. In this command, we are invoking python with the file `hello.py`. Python interpreter will read through the file and execute it on the fly. Try adding more `print` statements to the file and see the output.

In python, you need not put a semicolon to end a statement. Newline will be contextually identified as the end of the statement. Yet, to have multiple statements in the same line, you may use a semicolon.

```
1 print("Statement 1")
2 print("Statement 2"); print("Statement 3")
3 print("Statement 4")
```

Listing 2: "hello.py"

3 IPython

One of Python's most useful features is its interactive interpreter. It allows very fast testing of ideas without the overhead of creating test files. Issue command `python3` in your terminal to start the interactive interpreter. In the prompt that you get, try our "Hello World" print statements. You will see something like this:

```
user@host:~$ python3
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello World")
Hello World
>>> print("Hai")
Hai
>>> quit()
```

While this is very useful, the interpreter supplied with the standard Python distribution is somewhat limited for extended interactive use. IPython or "Interactive Python" provides a much more sophisticated interactive interpreter. Use `quit()` to exit the interpreter and try `ipython3` to start IPython.

```
user@host:~$ ipython3
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
Type "copyright", "credits" or "license" for more information.

IPython 5.5.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: print("Hello World")
Hello World

In [2]: quit()
```

Apart from the color coding that you notice, IPython has a lot of useful features that we will see soon.

4 Types and Operators

4.1 Basic Types

Python is a strongly typed language. It recognizes a variety of types but doesn't require the user to declare any of them explicitly. They include:

- Integers
- Floats
- Complex numbers
- Boolean values
- Strings
- Tuples and Lists (these are arrays)
- Dictionaries (these are associative arrays)

Python does not expose pointers to the user, all the memory management is handled invisibly. Python has the ability to define other types as well, such as matrices through the use of classes. Let's try some examples. Start IPython in your terminal and try these examples:

```
In [1]: a = 21   # define integer variable a and assign the value 21
In [2]: b = 2.1  # define float variable b and assign the value 2.1
In [3]: c = "21" # define string c and assign the value "21"
In [4]: c = '21' # You can use " as well as ' to define strings
In [5]: d = 10

In [6]: a/d
Out[6]: 2.1

In [7]: a//d     # floor division of a by d
Out[7]: 2
```

`# comment` is the syntax for comments in python. Notice here that we did not explicitly mention the types of variables. Python takes care of it for us.

```
In [8]: i = -3; j = 2;

In [9]: i//j
Out[9]: -2
```

Note that in this example, -2 is simply $\lfloor \frac{-3}{2} \rfloor$.

```
In [10]: z = 1 + 2j
```

Python handles complex numbers as native types. This is enormously useful to scientists and engineers which is one reason why it is popular with them. Note that `j` is identified as $\sqrt{-1}$ by context. So the following code will give peculiar answers:

```
In [11]: j = -2

In [12]: 1 + 2j
Out[12]: (1+2j)

In [13]: 1 + j
Out[13]: -1

In [14]: 1 + 1j
Out[14]: (1+1j)
```

In the first complex assignment(`1+2j`), Python knew that you meant a complex number. But in the second case, it thought you meant the addition of 1 and the variable `j`. Also, note that if you assign the answer to a variable IPython does not print the answer. But if you do not assign it, the value is printed out.

```
In [15]: v = True

In [16]: j > 2 or v
Out[16]: True
```

`True` and `False` (capitalized as shown) are the fundamental boolean values. They can be assigned to variables and they are also the result of logical expressions.

```
In [17]: "This is the %dth sentence of %s." % (4, "the text")
Out[17]: 'This is the 4th sentence of the text.'
```

This is a peculiar expression. We have seen strings already. But any string can be a printf string that follows C syntax. This must be followed by a % sign and a list of values. Note that this is the general string, and can be used anywhere, not just where printing is going to happen. Following is another (relatively new) way of creating strings:

```
In [18]: "This is the {}th sentence of {}.".format(4, "the text")
Out[18]: 'This is the 4th sentence of the text.'
```

`format()` function is capable of doing much more. You can explore it online.

```
In [19]: """This is a multi
```

```
    ...: line string"""
Out[19]: 'This is a multi\nline string'

In [20]: print(_)
This is a multi
line string
```

Sometimes we might want embedded newlines in strings. Then the above syntax helps us. Everything between the triple quotes is part of the string. Note the `print` command. It prints "_" which is Python's variable that holds the last computed value that was not assigned to a variable. Naturally, there is a large collection of built-in functions to work with strings.

Home work

Explore the following functions on strings:

1. `split()`
2. `count()`
3. `find()`
4. `isalnum()`
5. `isalpha()`
6. `strip()`
7. `zfill()`

```
In [21]: str(123)
Out[21]: '123'
In [22]: float("12.3")
Out[22]: 12.3
In [23]: int("abc")
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-12-2dda1cc00c48> in <module>()
----> 1 int("abc")

ValueError: invalid literal for int() with base 10: 'abc'
```

While data conversions are very simple with python, unexpected inputs will result in errors. We should "catch" these errors to make the program robust. We will see how to do the same in the next lab.

4.2 Operators

Python provides all the common operators like add, subtract and multiply. Note that operators for Boolean operations are not symbols but operator names themselves!

```
In [24]: 1 + 2
```

```
Out[24]: 3

In [25]: 1 == 2
Out[25]: False

In [26]: True or False    # Boolean OR
In [27]: True

In [28]: True and False   # Boolean AND
In [29]: False

In [30]: not False        # Boolean negation
In [31]: True

In [32]: 2**4             # 2^4
Out[32]: 16

In [33]: 2 in [1, 2, 3]   # Does [1,2,3] have element 2?
Out[33]: True

In [34]: a = 2; b = a; a is b
Out[34]: True
```

Note that `is` is different from `==`. For example:

```
In [35]: a = 2; b = 2.0; a is b
Out[35]: False

In [36]: a==b
Out[36]: True
```

4.3 Lists

Lists store items in order so that they can be accessed via their index. This is the python equivalent of C arrays. The indexing follows the C indexing (i.e., starts from 0). Unlike C, the elements of a list do not have to be identical in type.

```
In [37]: arr = [1, 2.1, 3, "string"];

In [38]: arr
Out[38]: [1, 2.1, 3, "string"]
```

Arrays can be created by just specifying the elements as in the first example above. The "constructor" is a set of comma separated values enclosed in square brackets.

```
In [39]: arr = range(10); list(arr)
```

```
Out[39]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [40]: arr = range(2,5); list(arr)
Out[40]: [2, 3, 4]
```

A common use for arrays is to create a sequence of uniformly spaced points. This can be done with the `range()` function. Without a second argument, it returns a sequence of integers starting with zero up to the given argument (but not including it). With two arguments, the sequence starts at the first argument and goes up to the second. Given three arguments, the third argument is the step. Note that range expects integer arguments.

One thing to notice here is the use of `list()`. It is simply doing a type conversion of output of `range()` to List (Similar to earlier examples of `str(), int(), float()` etc...). In python 3, the `range()` function does not return a pure list. Instead, it returns a sequence object. Sequence objects can be used to loop through the sequence it is representing. `list()` here converts the sequence object to a list. The ability to work with sequence objects or sequence generators is another powerful feature of python. In fact, the potential comes from python being lazy! In this case, since python is only creating the sequence object and not the sequence itself, it saves a lot of space. Moreover, the sequence is not computed until you use it, thus saving the computation time as well. See how laziness saves us space and time!

In IPython, `<entity>?` will give you the details of the entity. The entity here could be a function, variable, modules etc... (We will discuss about modules soon)

```
In [41]: range?
Init signature: range(self, /, *args, **kwargs)
Docstring:
range(stop) -> range object
range(start, stop[, step]) -> range object

Return an object that produces a sequence of integers from start (inclusive)
to stop (exclusive) by step.  range(i, j) produces i, i+1, i+2, ..., j-1.
start defaults to 0, and stop is omitted!  range(4) produces 0, 1, 2, 3.
These are exactly the valid indices for a list of 4 elements.
When step is given, it specifies the increment (or decrement).
Type:           type

In [42]: a = [1,2,3]

In [43]: a?
Type:        list
String form: [1, 2, 3]
Length:      3
Docstring:
list() -> new empty list
list(iterable) -> new list initialized from iterable's items
```

4.3.1 List indexing

The list indexing in Python is quite advanced compared to C.

```
In [44]: arr=[1, 2, [4,5,6], "This is a string"]

In [45]: arr[0]
Out[45]: 1

In [46]: arr[2]
Out[46]: [4, 5, 6]

In [47]: arr[3]
Out[47]: 'This is a string'

In [48]: arr = list(range(10)); arr[1:7:2]
Out[48]: [1, 3, 5]
```

We can refer to a portion of an array, using what is known as a slice. 1:7:2 means *start with index 1, increment by 2 up to but not including index 7.*

```
In [49]: arr[2][1]
Out[49]: 5

In [50]: arr[3][2]
Out[50]: 'i'

In [51]: arr[-1]
Out[51]: 'This is a string'
```

These syntaxes are quite intuitive if read correctly. `arr[2]` is a list. In `arr[2][1]`, you are then indexing the result of `arr[2]`. Visualize this as `(arr[2])[1]`. `arr[3][2]` is also similar. Note that in Python strings are index-able just like lists. The last command here is interesting. The negative number given is indexing the list backward. Note that there is no negative zero, so -1 refers to the last element.

Can you combine negative indexing with slicing? Try out all the combinations you can think of.

4.3.2 Homework

Refer and find

1. How to create an empty list
2. How to add/remove an element to/from a list

3. How to replace an element in the list
4. How to combine two lists
5. How to sort a list of integers.
   What happens if the list you sort has strings too?

There is another type of list in Python called the tuple. It merely means a constant list, i.e. a list whose values cannot change. It follows the same syntax except its values are enclosed in round brackets.

4.4 Dictionaries

Python also has associative arrays whose values are indexed not just by numbers but by any hashable objects. For now, let's not worry about what is a hashable object. Keep in mind that integers, floating point numbers, string etc... are all hashable. But a list is not hashable. Note that a tuple is hashable.

```
In [52]: hash = { "a": "Argentina", "b": "bolivia", "c": [1,2,3] }
         # initializing a dictionary

In [53]: hash["a"]
Out[53]: 'Argentina'

In [54]: hash["c"]
Out[54]: [1, 2, 3]

In [55]: hash[0]
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-27-df0d218756dd> in <module>()
----> 1 hash[0]

KeyError: 0

In [56]: hash[0] = "zero"

In [57]: hash
Out[57]: {0: 'zero', 'a': 'Argentina', 'b': 'bolivia', 'c': [1, 2, 3]}

In [58]: hash[0]
Out[58]: 'zero'
```

The first attempt of `hash[0]` fails since the elements of dictionary are not accessed by an index. The error message says that `0` is not a key found in the dictionary. A dictionary consists of an unordered set of key-value pairs. Each pair has an object(string, integer, etc...) called the key and some value. The value could be any object. In the above example string `"c"` is a key and the list `[1, 2, 3]` is the value corresponding to `"c"`.

5 Basic Programming

The basic structures in Python are the `if` block, the `for` block and functions. There are others, but these are the important ones.

5.1 `if` Blocks

```
In [59]: if j > 2 or v:
    ...:     print("Too many inputs: %d" % j)
    ...:
Too many inputs: 3
```

Note that blocks start with a statement ending in a colon. Unlike C, python does not use curly brackets to identify blocks. Instead, the extent of indentation is how python knows which block the statement belongs to. Thus the code in the block must be indented evenly. The above code is the standard if block. The general structure is

```
if condition:
    commands
elif condition:
    commands
elif condition:
    commands
...
else:
    commands
```

There is no switch command in Python, and this construction has to do the job of the switch-case statement as well.

5.2 `for` Loops

```
In [60]: sum = 0

In [61]: for  i in range(20):
    ...:     sum+=i
    ...:

In [62]: print("The sum of 0, 1, ... 19 = %d" % sum)
The sum of 0, 1, ... 19 = 190
```

The `for` block in Python iterates over an array of values. Here it is iterating over a set of integers.

```
In [63]: words = ["Water", "water", "everywhere", "but", "not", "a", "drop", "to"
    , "drink"]
In [64]: for w in words:
    ...:     print("%s" % w, end=" ")
    ...:
Water water everywhere but not a drop to drink
```

This for block prints out all the values of array `words`. Note that the `print()` function is given a second argument named `end` which replaces the carriage return at the end with the given string.

5.2.1 List Comprehension

Equipped with the knowledge of for loops, let's jump back to our lists for a while. Let's attempt to create some lists using for loops.

```
In [65]: l = []
In [66]: for i in range(4):
    ...:     l.append(2**i)
    ...:
In [67]: l
Out[67]: [1, 2, 4, 8]
```

Here, `**` is the operator for power. i.e. `2**i` is nothing but 2^i. While we are able to generate our list, this is not the pythonic way of doing it.

```
In [68]: l = [ 2**i for i in range(4) ]; l
Out[68]: [1, 2, 4, 8]
```

This method will also give the same result as before. Notice that this is short as well as more readable for a "Python-guy". This method of creating lists is called list-comprehension.

Home Work

1. Write corresponding for loop for the following commands:

   ```
   [ i for i in range(10) if i%2==0 ]
   [ str(i)+" is "+("even" if i%2==0 else "odd") for i in range(1, 5)]
   ```

   Note that `<value> if <condition> else <value>` is nothing but the ternary operator of python.

2. Write the command to generate prime numbers in single line. Don't worry about the efficiency of your command.

5.3 `while` Loops

```
In [69]: sum = 0; i = 0
In [70]: while i < 20:
    ...:     i += 1
    ...:     if i%3 == 0:
    ...:         continue
    ...:     sum += i
    ...:

In [71]: print("The sum = %d" % sum)
The sum = 147
```

The while construct is self-evident. It iterates till the condition fails. The `if` command skips to the next iteration every time a multiple of 3 is encountered. Also, note that the `continue` statement can be placed on the `if` line itself if desired. This is not actually recommended though.

5.4 Functions

```
In [72]: def factorial(n=0):
    ...:     """ Computes the factorial of an integer """
    ...:     fact = 1
    ...:     for i in range(1, n+1):
    ...:         fact *= i
    ...:     return fact
    ...:

In [73]: factorial(5)
Out[73]: 120

In [74]: factorial()      # use the default value of n
Out[74]: 1

In [75]: factorial(3.5)
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-62-24f344214769> in <module>()
----> 1 factorial(3.5)

<ipython-input-58-3f057ea44b00> in factorial(n)
      2     """ Computes the factorial of an integer """
      3     fact = 1
----> 4     for i in range(1, n+1):
      5         fact *= i
      6     return fact
```

```
TypeError: 'float' object cannot be interpreted as an integer
```

Functions are also defined using indented blocks. Arguments are like C and can have default values. Here the default value of `n` is zero, which is the value assumed for n if no argument is passed. If a float is passed, an error is generated. This is not an argument mismatch error - python is very forgiving about those. From the error message, it is evident python has triggered function call and has also executed the statements till line 4. The error has occurred only at line 4 while invoking `range()` function since it requires integer arguments.

The tripple quoted string under function name shows the self-documenting nature of Python functions. Anything entered as a triple-quoted string at the beginning of a function is retained as documentation for the function.

6 File Handling

There are many commands for file management. I will walk you through some of them. Reading up the remaining is your job. Let's create a dummy file named test.txt and add some text in it for testing purpose.

```
f = open("test.txt")
data = f.read()
f.close()
```

The first line opens the file for reading and assigns it to the file pointer `f`. The next line reads in the entire file into `data` as one long string. You should always close the file after reading it.

```
f = open("test.txt")
lines = f.readlines()
f.close()
print("1st line is %s" % lines[0])
```

`f.readlines()` will return a list of strings corresponding to each line in the file.

A more "pythonic" way of doing the above tasks is by using `with` keyword.

```
with open("test.txt") as f:
    lines = f.readlines()
    print("1st line is %s" % lines[0])
```

Here, `open()` is executed and the result is aliased with `f`. Using `with` will free you from things like closing the file. It will be Python's responsibility to create and destroy the file streams in a clean manner.

```
In [76]: with open("file.txt", "w") as f:
    ...:     f.write("First line\n")
    ...:     f.write("Second line\n")
    ...:     f.write("Last line\n")
```

This is the syntax to write to a file. You can replace `"w"` with `"a"` for appending to the file.

7 Modules

Python comes with a huge variety of helper functions bundled in modules. For now, all we need to know is that modules are collections of functions that have their own "namespace". You import a module by using the import command:

```
In [77]: import os
```

Now that we have imported the module `os`, we can use the functions defined in it. `system()` is a function defined under the module `os`. Read about both the module as well as the function using `?` symbol in IPython.

```
In [78]: os.system?
Signature: os.system(command)
Docstring: Execute the command in a subshell.
Type:      builtin_function_or_method

In [79]: os.system("ls")
Out[79]: # This output will be specific to your computer.
```

The command `os.` is helping us to use the functions that are defined in the namespace of `os`. `ls` is the bash command used to list contents of a directory in Linux.

You can also import all/specific functions inside a module using the following syntax:

```
In [80]: from os import *       # import everything inside os
In [81]: from os import system  # import function system() from module os
```

You may visualize this hierarchy like a file-system. Think about `os` as a directory and `system()` as one inside it.

7.1 Commandline arguments

IPython is only an environment where you can test your ideas. Large projects need to be written in a file and executed using the command `python <file>`. Sometimes, we need to

supply some arguments to the program in the commandline. You should be already familiar with this from C.

In Python, we take the help of `sys` module to access the commandline arguments.

```
1 import sys
2 arg0 = sys.argv[0]
3 arg1 = sys.argv[1]
4 print("number of arguments received by %s = %d" % (sys.argv[0], len(sys.argv)))
```

8 Exception Handling

Exceptions are events that occur as a result of unexpected cases in code. For example,

```
In [82]: age = input("What's your age? ") # This is how to accept input
What's your age?  2.3

In [83]: type(age)            # This is how to find type of a variable
Out[83]: str                  # age is string data type

In [84]: age = int(age)       # Let's convert the age to int data type
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-4-9d45e9e78105> in <module>()
----> 1 age = int(age)

ValueError: invalid literal for int() with base 10: '2.3'
```

Here, I did not foresee that the user might enter a value which cannot be converted to integer. The `int()` function is **raising** a ValueError which is causing my program to crash. The right way to tackle this problem is to validate user input and make sure that it can actually be converted to integer before you invoke `int()` function. For example, `age.isdigit()` will return `True` if `age` represents a whole number.

Another approach would be to get alerted when the function is complaining about an error. We shall call this complaint as exception. We, getting alerted about the exception is called catching the exception.

```
In [85]: try:
    ...:     age = int(age)
    ...: except Exception:
    ...:     print("Invalid age")
Invalid age
```

As soon as the code inside `try` block causes an exception, the control is transferred to the code inside `except` block. You can now handle the error case here. I have simply printed an error message, but you can do better. Note that the program is not crashing in this case.

There are many kinds of exceptions of which `Exception` is the super set of all of them. If you read through the error message from `int()` carefully, you will see that it says `ValueError`. This is a subset of `Exception` (For those of you familiar with OOP concepts, `ValueError` is inherited from `Exception`). It's a good practice to use the smallest subset instead of using `Exception`.

```
In [86]: try:
    ...:     age = int(age)
    ...: except ValueError as e:
    ...:     print(e)
invalid literal for int() with base 10:  '2.3'
```

The raised `ValueError` is caught and aliased as `e` here.

9 Spice - Part 1

I think the best way to learn a language is to do a project in it. So, we will attempt to build a slightly complex program in python and learn the language on the way. Let's try and build a spice program in python. We will learn more about python on the way.

9.1 Introduction

In this experiment, let's attempt to write a program that can solve electrical circuits for us. We will build this in 3 parts. This week, we will create the first part where we will read the input file to construct the circuit. The circuit has only simple devices. Over the coming weeks, we will analyze the input that was read and solve the circuit.

9.2 Spice netlist

A spice netlist consists of lines of the following forms:

```
....
.circuit
name n1 n2 value # comment
name n1 n2 n3 n4 value # comment
name n1 n2 vname value # comment
.end
```

```
....
```

When reading, we are to neglect lines till we meet a dot command, i.e., a line starting with a dot. If the dot command is `.circuit`, the following lines define the circuit. The circuit definition ends with a line containing `.end`. Note: These dot commands must have `.` in the first character of the line. This distinguishes them from lines where a `.` comes in the middle of the line.

The circuit definition consists of lines, each of which defines a branch of the circuit. For instance,

```
name n1 n2 n3 n4 value # comment
```

is a branch representing a dependent source, where `n1`, `n2` , `n3` and `n4` are symbolic names representing nodes, name is the name of the element. The first letter of the name defines the type of element, as follows:

`name`	Description	Units	Definition
`R....`	Resistor	Ω	`R.... n1 n2 value`
`L....`	Inductor	H	`L.... n1 n2 value`
`C....`	Capacitor	F	`C.... n1 n2 value`
`V....`	Independent Voltage Source	V	`V.... n1 n2 value`
`I....`	Independent Current Source	A	`I.... n1 n2 value`
`E....`	Voltage Controlled Voltage Source	V	`E.... n1 n2 n3 n4 value`
`G....`	Voltage Controlled Current Source	A	`G.... n1 n2 n3 n4 value`
`H....`	Current Controlled Voltage Source	V	`H.... n1 n2 V.... value`
`F....`	Current Controlled Current Source	A	`F.... n1 n2 V.... value`

`value` is the value of the element in Ω, H, F, Amps or Volts, respectively. In the case of current controlled sources, the controlling current must be the current through a voltage source. This is not a limitation since all we need to do is to add a zero volt voltage source where we want and use its current as the controlling value.

Incidentally note that zero volt voltage sources are frequently used in spice to measure current in a branch of interest. This is because the equations automatically solve for the current through voltage sources so that they become available for further analysis.

The line can optionally be followed by a comment. This is defined as anything that follows a `#` character.

Each impedance line in the file translates into an equation:

$$V_{n1} - V_{n2} = I_{n1,n2}\, R_{n1,n2}$$

$$V_{n1} - V_{n2} = L_{n1,n2}\, \frac{dI_{n1,n2}}{dt}$$

$$I_{n1,n2} = C_{n1,n2}\, \frac{d(V_{n1,n2} - V_{n1,n2})}{dt}$$

Each source line in the file becomes

$$V_{n1} - V_{n2} = \mathcal{E}_{n1,n2}$$

$$I_{n1,n2} = \mathcal{I}_{n1,n2}$$

$$V_{n1} - Vn2 = E(V_{n3} - V_{n4})$$

$$V_{n1} - Vn2 = FI_k$$

$$I_{n1,n2} = G(V_{n3} - V_{n4})$$

$$I_{n1,n2} = HI_k$$

where I_k is the current through the controlling voltage source.

9.3 Assignment

Write the pseudo code to do the following

1. Accept the name of netlist file as commandline. argv
2. Check if the user has actually provided the filename or not and give appropriate error message if needed.
   It is always a good practice to validate the number of arguments given to the program. Note that a good program never crash. It will check for all the carelessness that can happen with the input and take necessary actions.
3. Determine the section that contains the circuit definition (i.e., the portion that started with a `.circuit` line and ended with a `.end` line)
4. Parse each line of the section and extract the words (tokens). split()
5. Analyze the tokens and determine the from node, the to node, the type of element and the value. If it is a dependent source, you also need to extract additional information. Note that node names are alphanumeric.

6. When all the lines have been read in, close the file.

7. Traverse the circuit definition **from last element to first** and print out each line with **words in reverse order**.
   Eg: if the netlist is

```
.circuit
R... n1 n2 value
E... n1 n2 n3 n4 value
.end
```

   Print

```
value n4 n3 n2 n1 E....
value n2 n1 R...
```

Now write the code to do the same in Python. You should submit this code on Moodle.