

Programming, Data Structures and Algorithms in Python
Prof. Madhavan Mukund
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week - 01
Lecture - 04
Downloading and installing Python

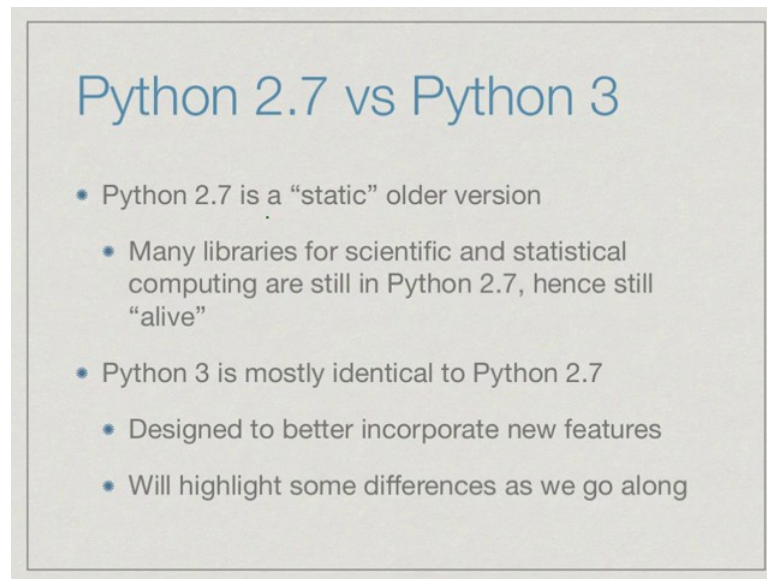
For our final lecture of this first week, we will see how to actually use Python on our system.

(Refer Slide Time: 00:10)



Python is a programming language, which is available on all platforms. So, whether you are working on Linux or on a Mac or on Windows, you will be able to find a version of python that works on your system. One of the small complications with python is that there are two flavors or two versions of python, which are commonly found. So, there is an older version called python 2.7, and there is a newer version called python 3. Python 3 is a one that is being actively developed, python 2.7 is more or less a static version and currently python 3 has the version 3.5.2 or something like that. So, there is not much difference whether you are using 3.5 or 3.4, but there are difference between 2.7 and 3. And for the purpose of this course, we will work with python 3.

(Refer Slide Time: 01:06)



What is the difference between these two versions? Well, python began with a few features and it kept developing into more versatile programming language. So, python went through much iteration and python 2.7 was a version that was reached when the developers of python decided that there should kind of make a clean start. And some of the new features which had been added in an ad hoc way on to the language should be integrated in a better way which makes it a more robust programming language.

Python 3 essentially is a modern version of python, which incorporates features that were added on to python as it grew in a way that makes it more consistent and more easy to use, but as often happens a lot of people had already been using python, and python 2.7 has a lot of software **written** using that version. In particular a lot of software that people find convenient to use such as scientific and statistical libraries of functions where they do not have to use it themselves, **they'll** just **invoke** these libraries **are** still **written** using python 2.7. And if you run it from python **3** sometimes these functions do not work as they are expected.

So, this has forced python 2.7 to **live** on. Eventually we hope that somebody will take the effort to move python 2.7 libraries to python 3. And of course, newer code is largely being developed on python 3, but you should remember that when somebody says that

they are using python they could be talking about 2.7 and not 3, and you have to make adjustments.

For the purpose of the introductory material that we will be doing in the course, there is almost no change between python 3 and python 2.7; however, there are some features that we will see which are slightly different in 2.7 and we will explore them in 3, and I will try to highlight these differences as we go long. But going forward in python 3 is the current version and it has been the current version for some years now at least for 4 or 5 years. It is definitely the language, which is going to dominate in the future, so it is better that you start with a new version then go back to the old version.

(Refer Slide Time: 03:23)



The slide is titled "Downloading Python 3.5" in a large, blue, sans-serif font. Below the title, there are four bullet points, each preceded by a blue asterisk. The text is in a smaller, grey, sans-serif font. The slide has a light grey background with a thin black border.

- Any Python 3 version should be fine, but the latest is 3.5.x
- On Linux, it should normally be installed by default, else use the package manager
- For MacOS and Windows, download and install from <https://www.python.org/downloads/release/python-350/>
- If you have problems installing Python, search online or ask someone!

As far as this course is concerned, any version of python 3 should be fine. The latest version as I said is some 3.5.x, where I think x is 2, but if you do not have 3.5, but you have 3.4 or 3.3 do not bother everything should work fine. But if you are interested, you can install the latest, latest version. If you are using Linux, it should normally be there by default because many Linux utilities require python and so python should be on your system, but it could be that the utility is using python 2.7. So, make sure that you install python 3. You can use the package manager to do this. Now if you are using a MAC or

you are using Windows then python may or may not be installed especially python 3 may not be installed on your system.

There is the URL given here. If you search on Google, you will find it. Just search for python 3.5 install or download and you will get to this URL. So, www.python.org/downloads/release/python-350. 350, is really refer into 3.5.0. So, actually the current version as I said is not 3.5.0, but 3.5.2. So, you will find instructions there - please download the version that is appropriate for your system and install it. These are **designed** to be fairly self-explanatory install files; if you have a problem please search online for help with the problem you are facing or ask someone around **you**. **It** is not the purpose of this course to spend a lot of time telling you how to install software. So, I hope you are able to do this, so that we can get ahead with the actual programming part.

(Refer Slide Time: 05:05)

The slide is titled "Interpreters vs compilers" in blue text. It contains four bullet points. The first bullet point says "Programming languages are 'high level', for humans to understand" with the handwritten note "Arrange chairs" next to it. The second bullet point says "Computers need 'lower level' instructions" with the handwritten note "Put 80 chairs in 8 rows, 10 each" next to it. The third bullet point says "Compiler: Translates high level programming language to machine level instructions, generates 'executable' code". The fourth bullet point says "Interpreter: Itself a program that runs and directly 'understands' high level programming language".

Interpreters vs compilers

- Programming languages are "high level", for humans to understand "Arrange chairs"
- Computers need "lower level" instructions "Put 80 chairs in 8 rows, 10 each"
- Compiler: Translates high level programming language to machine level instructions, generates "executable" code
- Interpreter: Itself a program that runs and directly "understands" high level programming language

One more thing to keep in mind, if you are familiar with other programming languages, is the distinction between interpreters and compilers. So, the main difficulty is that programming languages like python or C or C++ or Java are written for us to understand and write instructions on. So, these are somewhat high level instructions. In the other hand, computers need low level instructions. So, when we talk about names and values like i, j or we talk about list, **the** underline computer may not be able to directly analyze

these things, so we need a translation. If you remember the very first lecture, we talked about arranging **chairs**. So, we said arrange the **chairs** as a high level thing, and we said put 80 chars in 10 in 8 rows, 10 each right.

We said that there could be a difference in the level of detail in which you give instructions and this is precisely what happens. In order to execute something so called executable file that we come across we have something which is written at a level that the machine can understand. Whereas, the programs that we are going to explore on this course and which all programmers normally work with are at a higher level, which cannot be directly understood by computer, so we have to bridge this gap somehow.

A compiler is a program which takes a high level programming language and translates programs written in that language to a machine level programming language. So, it takes the high level program in python, **if** in not python, in C or C++ or Java or something and produces something which directly a machine can execute. In the other hand, the other way of dealing with the high-level language is to interpret it. So, an interpreter in normally English is somebody who stands between people talking different languages and translates back and forth.

An interpreter is a program which you interact with, and you feed the interpreter instructions in your language, in this case python; and the interpreter internally figures out how to run them on the underlying machine. So, whether you are running it on Windows, or Mac, or Linux **interpreter** guarantees that the answer that you see at the high level looks approximately the same independent of the actual platform on which you are running it. So, python is by and large an interpreted language and we should be aware of this fact.

(Refer Slide Time: 07:31)

Python interpreter

- Python is basically an interpreted language
 - Load the Python interpreter
 - Send Python commands to the interpreter to be executed
 - Easy to interactively explore language features
 - Can load complex programs from files

```
>>> from filename import *
```

filename.py

We use python typically in the following way; we first run the interpreters. So, remember interpreter is the program. We first invoke the interpreter; and when the interpreter is running, we pass python commands to the interpreter to be executed. The nice thing about dealing with an interpreter is that you can play with it like you play with a calculator; you can feed it commands and see what it does, so it is very **interactive**. Of course, it is tedious, if you have to type **in** large **programs**, so there is a way to load a program which has been written already using a standard text editor and loading it from a file. So, what I have shown below in green is so this is what we will see in a minute is the prompt **that** the interpreter shows you.

When you enter the interpreter, it will ask you to execute a command and this is a command that you provide the interpreter. It says. So, I have stored. I have a file called say file name dot p y typically **to** indicate **it** is a python program from that file import all the definitions and functions and code that is **written there**. So, this will tell the interpreter to take everything that is written in that code and put it into **its** current **environment**, so that those functions can be used. So, these things will become a little clear and then in the demo that I am just going to show you and then you can play around with this. And then the next week, we will get into the real details about exactly what goes into a python program.

(Refer Slide Time: 09:16)

```
$ ls
__pycache__/  gcdeuclid1.py  gcdeuclid1a.py  gcdeuclid2.py  gcdeuclid2a.py
gcdfour.py    gcdone.py      gcdthree.py     gcdtwo.py
$ python3.5
Python 3.5.2 (default, Jun 27 2016, 03:10:38)
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> i = 5
>>> i
>>> 5

>>> i+1
>>> 6

>>> 2*3 + 5
>>> 11

>>> def twice(x):
...     y = 2*x
...     return(y)
...
>>> twice(7)
>>> 14

>>> twice(932)
>>> 1864

>>> quit()
>>> $
```

Here is a window showing the terminal which on Windows would be like a command prompt and using unique like shell. So, if I say ls, it shows me the list of files in my current area. And all this files with extension dot p y are actually python programs. In this, I invoke the python interpreter by saying python 3.5 because that is the **version** which I am using. If I invoke it, it will produce some messages telling me what type of function system **I'm on**. So, it tells me that I am using for instance 3.5.2 and it has may that it is a fairly recent version, it tells me that it is on a Apple and blah blah blah, but what is important is then produces a prompt place where I can enter commands and this is signified by these three greater times.

Now, at the python interpreter prompt, you can directly start writing things. So, for example, you can say i is equal to 5. What **it** says as a take a name i assign to value of 5. Now if I **type** i, it tells me that the **value is** 5; if I type an expression like i plus 1, it tells me that is 6. So, you can use it as a calculator. So, you can do simple arithmetic if you want. So, you can keep interacting with it. Now, you can also define functions remember how we defined a function, we use def, use a function **name** and so on. So, we can say for example, def twice x. This is the function twice, this takes the single argument x. And as you might expect I would like to **return** two times x.

Now a python uses as we mentioned in one of the earlier lectures, indentation in order to specify that something is a part of something else. So, the definition consists of a bunch of it steps. So, I must tell it that these bunches of steps belong to this definition by indenting it; it does not matter how you **indent it** as long as you use the same indentation uniformly. If you are using two spaces, use two spaces use a tab, but do not mix up the number of spaces and do not mix up tabs and spaces, because this gets you confuse the **error messages** form python. So, let us use two spaces.

Let us to the sake of illustration create a new name y, and say y is two times x. Now it is still continuing to ask me for the definitions, so the prompt has change to dot dot dot. Now I must induct it a same way and say return y. So, what I have done is I say this function takes in value x, computes two times x, and stores it in the name y, and returns the value of the name y, right. Now, when I am done with this, I give a blank line and this function is now defined. Now, twice 7 makes sense, what twice 932 will also **make sense** right. So, python is very convenient in that you can have few define functions as you go along on the fly.

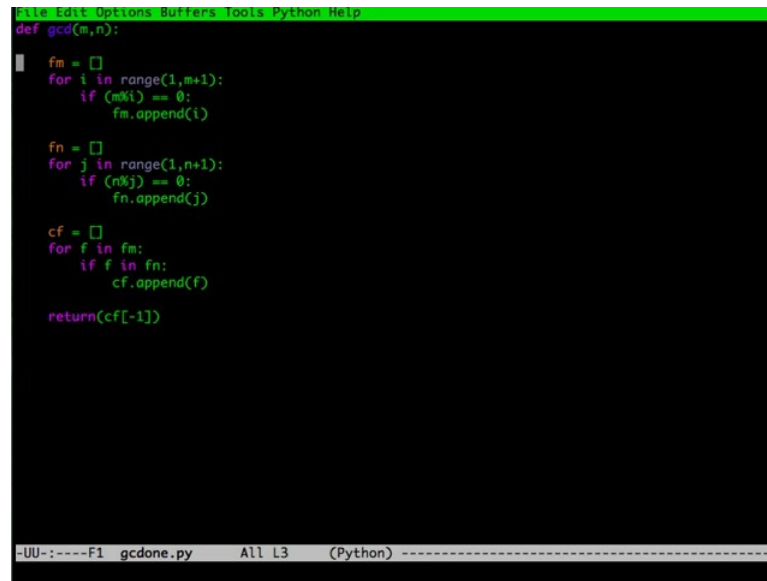
Now, we could also define our gcd right here, but as you might expect sometimes a function is too **complicated** to typing without make in a mistake, and secondly, you might want to play around with the function and change it and not have to keep typing it again and again. For this, what we need to do is first type the function in to a file and then load the file here. Let us get out of this. So, one way to get out of this is to type quit the brackets.

(Refer Slide Time: 12:49)

```
$  
$  
$  
$ ls  
__pycache__/  gcdeuclid1.py  gcdeuclid1a.py  gcdeuclid2.py  gcdeuclid2a.py  
gcdfour.py    gcdone.py      gcdthree.py     gcdtwo.py  
$ emacs gcdone
```

And then you get back to this prompt which is dollar which is the outside terminal or the command prompt. So, I have actually already created something. Let us start with, so I use an editor called emacs, you can use any takes editor if you are using Windows, you can use notepad, if you are using and Linux, you can use emacs or vi or you can use some simpler editor like gedit or k, anything that is **comfortable**, but it should just be a text editor it should not do any formatting, do not use word processes like you know office or something like that. You something we just manipulates text files.

(Refer Slide Time: 13:27)



```
def gcd(m,n):  
    fm = []  
    for i in range(1,m+1):  
        if (m%i) == 0:  
            fm.append(i)  
  
    fn = []  
    for j in range(1,n+1):  
        if (n%j) == 0:  
            fn.append(j)  
  
    cf = []  
    for f in fm:  
        if f in fn:  
            cf.append(f)  
  
    return(cf[-1])
```

If I look at gcd 1 dot py, so one nice thing what emacs is **it** shows me colors to indicate certain things. So, def this is the very first gcd program we wrote, which takes computes the list f m then the list f m then the list cf, and then it returns the last elements in cf. So, this is the first version of gcd. So, this is the exactly the code we wrote before. The point to remember is that I have made sure that all these indentations are at the same number of spaces in. So, this is something to remember. Now, you typing something like this right then you save it and exit.

(Refer Slide Time: 14:03)

```
$
$
$
$ ls
__pycache__/  gcdEuclid1.py  gcdEuclid1a.py  gcdEuclid2.py  gcdEuclid2a.py
gcdfour.py    gcdone.py      gcdthree.py     gcdtwo.py
$ emacs gcdone.py
$ python3.5
Python 3.5.2 (default, Jun 27 2016, 03:10:38)
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>> from gcdone import *
>>> gcd(14,63)
>>> 7

>>> gcd(999999,100000)
>>> 1

>>> gcd(999999,1000000)
>>> 1

>>> gcd(999999,1000000)
>>> 1

>>> quit()
>>> $
```

Now you go back to your python, and you save from that file `gcd 1 import star` what this means is take the file `gcd1 dot py` and load all the functions which had **defined** there and make them available to me here. Now, if I say `gcd` of 7 comma let us for example, 14 and 63 for instance, it tells me the `gcd` 7. Now if you take some large number like 9999 and 10000 then it takes, so may be one more digit let us see, you will notice that it is not giving me an answer and then it gives me answer. So, it this is just to illustrate that this was the slow `gcd` right. So, see how much time it took.

It has the visible gap of a few seconds before it produces the answer. And this is the illustration that this is not a very efficiency `gcd`. So, one of the problems with this python interpreter which I will see if we can solve is that if I have already loaded one file then it is safer to exit and then reload other file rather than to update the file.

(Refer Slide Time: 15:25)

```
$  
$  
$ ls  
__pycache__/  gcdeuclid1.py  gcdeuclid1a.py  gcdeuclid2.py  gcdeuclid2a.py  
gcdfour.py    gcdone.py      gcdthree.py     gcdtwo.py  
$ emacs gcdeuclid2.py
```

Let me reload for instance the last version of **Euclid's** thing, which we wrote which is the remainder version.

(Refer Slide Time: 15:32)

```
File Edit Options Buffers Tools Python Help  
def gcd(m,n):  
    if m < n:  
        (m,n) = (n,m)  
    if (m % n) == 0:  
        return(n)  
    else:  
        return (gcd(n,m%n))  
  
-UU-:----F1 gcdeuclid2.py All L1 (Python) -----  
For information about GNU Emacs and the GNU system, type C-h C-a.
```

It says that if m less than n exchange the values if then the second line here says that if the remainder of m divided by n is 0 that is n is a **divisor** of m then return n otherwise

replace the `gcd` call by the call to `n` and its remainder. So, this we also had a version of this **where** we return to the while loop. Let us use the while version. The while version says that so long as the remainder is not 0, we keep updating `m` and `n` to `n` and the remainder, and finally you return the value of `n`.

(Refer Slide Time: 16:13)

```
$  
$  
$ ls  
__pycache__/  gcdEuclid1.py  gcdEuclid1a.py  gcdEuclid2.py  gcdEuclid2a.py  
gcdfour.py    gcdone.py      gcdthree.py     gcdtwo.py  
$ emacs gcdEuclid2.py  
$ python3.5  
Python 3.5.2 (default, Jun 27 2016, 03:10:38)  
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwin  
Type "help", "copyright", "credits" or "license" for more information.  
>>> from gcdEuclid2a import *  
>>> gcd(9999999,1000000)  
>>> 1  
  
>>> gcd(9999999999,10000000000)  
>>> 1  
  
>>> bloop(7)  
>>> Traceback (most recent call last):  
      File "<stdin>", line 1, in <module>  
NameError: name 'bloop' is not defined  
  
>>> 7 < 5  
>>>      File "<stdin>", line 1  
      7 < 5  
        ^  
SyntaxError: invalid syntax  
  
>>> []
```

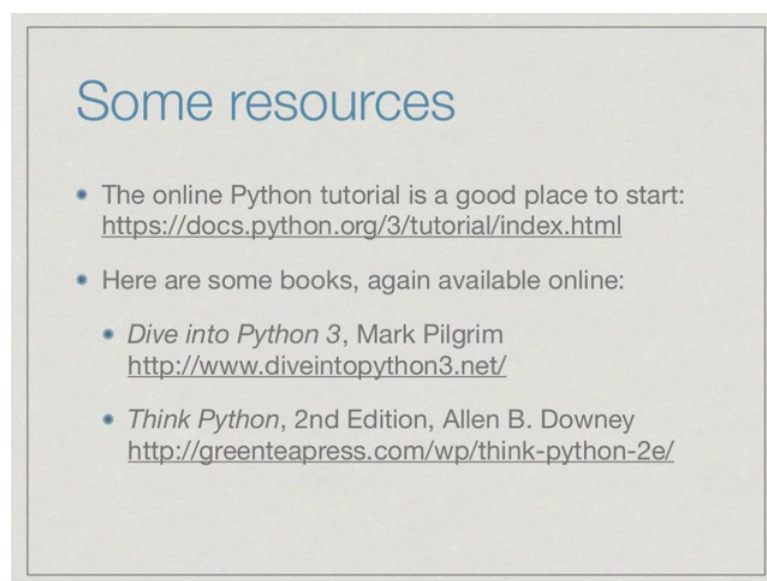
I am going to take this particular thing and load it into python. So, again I first invoke the interpreter python then I say `from gcdEuclid2a import star`. Now I am going to give that same large value that we saw before and which I think was say 9999999 and 1000000. And now you see, you get an instant answer. In fact, you will see that if I even if I give it several more digits, it should hope fully work fast. So, there is a dramatic improvement in speed which is even visible in this simple example, if we replace the naive idea by a clever idea.

The power of algorithm is to actually make a program which would otherwise be hopelessly slow work at a speed which is **acceptable to you**. Do a load python on your system, invoke the python interpreter and play around with the code that we have seen in this particular week's thing, make errors see what python tells you when you import a file which has errors. For instance now if I **try** to ah invoke a function which does not exists like, if I use a function which I have not defined and which python does not understand

then it will give me a mistake like this. It will say loop is not defined. If I write something strange like 7 less than greater than 5, then it will say that this is invalid syntax.

The interpreter will look for an expression if the expressions do not make sense then it is going to complain. And sometimes the error messages are easy to understand, sometime they are less easy to understand; as we go along we will look into this. But, the purpose of the interpreter is to either execute what you have given it or tell you that what you have written is somehow not executable and explains why. So, do play around with it and a get some familiarity because this is what going to be our bread and butter as we go along.

(Refer Slide Time: 18:18)



The slide is titled "Some resources" in a blue font. It contains a bulleted list of resources for learning Python. The first bullet point mentions the online Python tutorial with a URL. The second bullet point mentions books available online, with two specific examples: "Dive into Python 3" by Mark Pilgrim and "Think Python" by Allen B. Downey, each with a URL.

Some resources

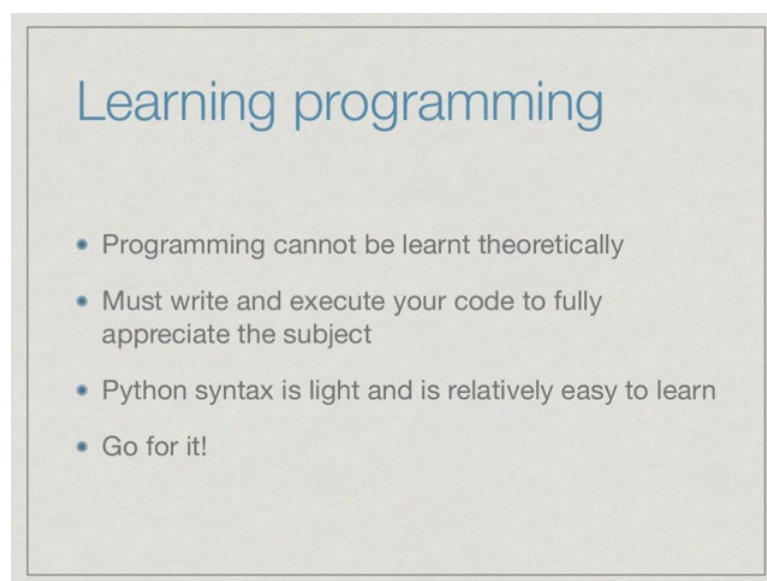
- The online Python tutorial is a good place to start:
<https://docs.python.org/3/tutorial/index.html>
- Here are some books, again available online:
 - *Dive into Python 3*, Mark Pilgrim
<http://www.diveintopython3.net/>
 - *Think Python*, 2nd Edition, Allen B. Downey
<http://greenteapress.com/wp/think-python-2e/>

We are going to be looking at some specific features of python in this course, but you may find as we go along that there is something that you do not understand or something new that you would like to try out your own. So, it is always a good idea to have access to other resources. The python online documentation is actually an excellent place to look for details about python and in particular, there is a very readable tutorial; especially, if you already have some familiarity with programming the python is probably the best place to start learning python for yourself. So, here is a URL,

docs.python.org/3 this is for python 3 tutorial index dot html. If you just go to docs.python.org/3, you will find there are also more detailed reference manuals and so on, which you might need at a later stage.

Do keep this as one of the places that you look when you have difficulties. And there are two books which probably useful to understand python beyond what is covered in the lectures if you feel that something is not clear. So, there is this book called dive into python which is **adapted** for python 3. And there is book called think python which is about generally about computational thinking in the context of python. Both of these have the nice advantage that they are available online, **so you** do not have to buy anything; you can just browse them through your **browser on** the net.

(Refer Slide Time: 19:41)



Before we **leave you** for this week, remember that learning programming is an activity; you cannot learn programming theoretically. You have to write and execute code to appreciate the subject. You have to make mistakes; learn from your mistakes; figure out what works, what does not work and **only** then will you get a true appreciation for programming. Reason we are going with python is because python has a very simple syntax **compared** to other programming languages. We have already without formally learning python, seen some fairly sophisticated programs **for** gcd and hopefully you have

understood them even if you cannot generate them. It is not very difficult to explain what a python program is doing with a little bit of understanding.

Do take the time to practice the examples that we had seen this time. We will be giving programming exercises as we go along; and unless you do these exercises and become somewhat handy at manipulating python yourself, you will never truly learn both programming and python. The other thing to remember is that once you have **learned** one language, even though the features and the syntax vary from language to language, it is very easy to pick up a another language, because all of programming has at it is base very similar principles.

Although the syntax may vary, the ideas do not. The ideas are eventually what write the program, but to be a fluent speaker of a programming language, you must practice it. So, do try.