2008

Data types, structures and algorithmic design

Victor Miclovich

The longer you stay with problems, the smarter you get...

Lecture 2
Data Structures, Types, and Basic Algorithm Design

Victor Miclovich

Prerequisites:

- Read chapters one through three to better understand this lecture; I write with the assumption that you have some fair knowledge on the subject matter
- Refer to a calculus text (applied math; pure I or II) and read about the Newton-Raphson method. Or better yet, read about numerical analysis; stuff to do with the Taylor's method, linear approximations, Iterations.
- A web search can be helpful, I advise that you stick to websites that don't use a lot of Flash media, but are mostly text based – HTML.

Course admin contacts:
vicmiclovich@gmail.com
v2kisitu@gmail.com
vic-the-doctor@hotmail.com
vkisitu@cit.mak.ac.ug
Wikiversity
Lecturer: Victor Miclovich
Course: Software Engineering →welcome to the virtual world, lol ☺.
If you run a google search with the above string, you might get a clear URL, or just link up to it. I am developing certain work (though basic) on software engineering

Preamble

Why programming is relevant?
Programming is a way we solve problems on earth. Our problems can be broken down to other smaller problems which can be further broken down and handled more easily.

Programs are used to do many things; ranging from cryptography (the art of concealing information from an adversary) to communications (protocols designed to enable inter-device "talk"), to a successful traversal through the orbit by a sate light.
Back on earth, accountants, engineers, doctors (list is practically endless) employ products of the creative genius of programming.

Introduction

Basic overview of a computer

I strongly believe that I should give you an overview of how a computer (or many non-QCs[1]) works.
First, a computer can be thought of as a device composed of three basic components: a **CPU**, an **I/O** device, and a **memory**. What I have just mentioned is commonly referred to as the von Neumann model.

A **CPU**, central processing unit, is the brains behind the operation of the computer. It orchestrates various computations that could be either logical or arithmetic. A CPU has some form of “primitive” memory we shall refer to as registers (a program counter and a host of others), an **A**rithmetic **L**ogic **U**nit for algebraic computations, a **B**us **I**nterface **U**nit to be able to communicate with the various components attached to the “system unit”.

The **I/O** devices are really just input (for putting data into a PC) devices and output (for bringing out information stored within) devices. Think of them as the interface between us (users) and the outside world.

Memory is what the machine uses to **cache** (store) operational data from a storage device **(I/O**) like a hard disk for the computers effective operation. The **CPU** picks instructions from main memory and executes them.

For all the above components to work effectively there has to be some form of interconnection. The interconnections I am referring to are just electric conduits for data in form of **digital signals**. A computer operates solely on digital signals (either 1 or 0, binary representation of data).
In a more advanced tone, I would like to say that these signals won’t appear as ones or zeros physical; you won’t see a one swimming its way through a way in a wire.
The primitive construct of memory is a switch (a transistor) that is set to function in either of two states of the binary values one and zero {0, 1}.

Buses are the electrical conduits I referred to a little bit earlier in these notes. Buses are just wires that the computer uses to communicate with other electronic components. If you ever open up a large electronic device like a Personal Computer, these buses are usually those white or gray cables that are the visible inter-device connections… you will see them all over the damn place!

[1]**QC**: Quantum computer

Mother/system board… this refers to the main electronic circuitry of the computer. The **CPU** is fixed onto it, the power from the computer's power supply is attached to it and all the buses have got certain connections to it (e.g. PCI, ISA)
This green board is printed and fabricated to contain various electronic components ranging from small minute resistors to transistors, capacitors and Integrated Circuits (**IC**) which are small microcontrollers. All this is like one huge orchestra playing a great symphony… in other words, this all works together to produce a perfectly playing system.

Memory Hierarchy

CPUs work on what is known as a *fetch-and-execute* loop (or cycle). Pieces of information are picked from the main memory and brought to the **CPU** to be interpreted and executed.

When you look at the trends of silicon-based technologies, each year the complexity in Processor design rises. Years of research and development has led to processors coming with amazing power and speed.

In the past, we were quite comfortable with 400 **MHz** processors (less than 1 **GHz** of speed) and 1 **MB** of **RAM**… this soon became obsolete as newer and speed-thirsty software applications were being developed and hence the speed had to be stepped up.

The Internet has also had a great impact on the development of this kind of technology! The need for large mainframes to work faster than the average PC, servers working effectively and using parallel processing to couple the powers of multiple processors to work with the ever increasing data speeds.

Speed has a negative impact on data-storage methods. A fast processor could mean the rate at which it works (executes various instructions) will be much faster than the rate at which data is being collected from the hard disk and shipped off to main memory, for example. So what systems designers did to work on this problem was to create memory hierarchies.
A memory hierarchy in effect would cache the data in steps to enable effective fetching of that data. (Storage-device technologies still are yet to be improved). An example of a memory hierarchy is shown below, Fig 1.00.
Fig. 1.0 illustrates the memory hierarchy. The top most level (**reg** → registers) is 0, and these layers down to the 6th are marked starting at zero (0) through five (5)

Data storage is of paramount importance to electronic ("intelligent") devices. And the lower echelons of storage (Level 0 through 5) always interface with each other to produce a close to smooth effect during a computer's operation.

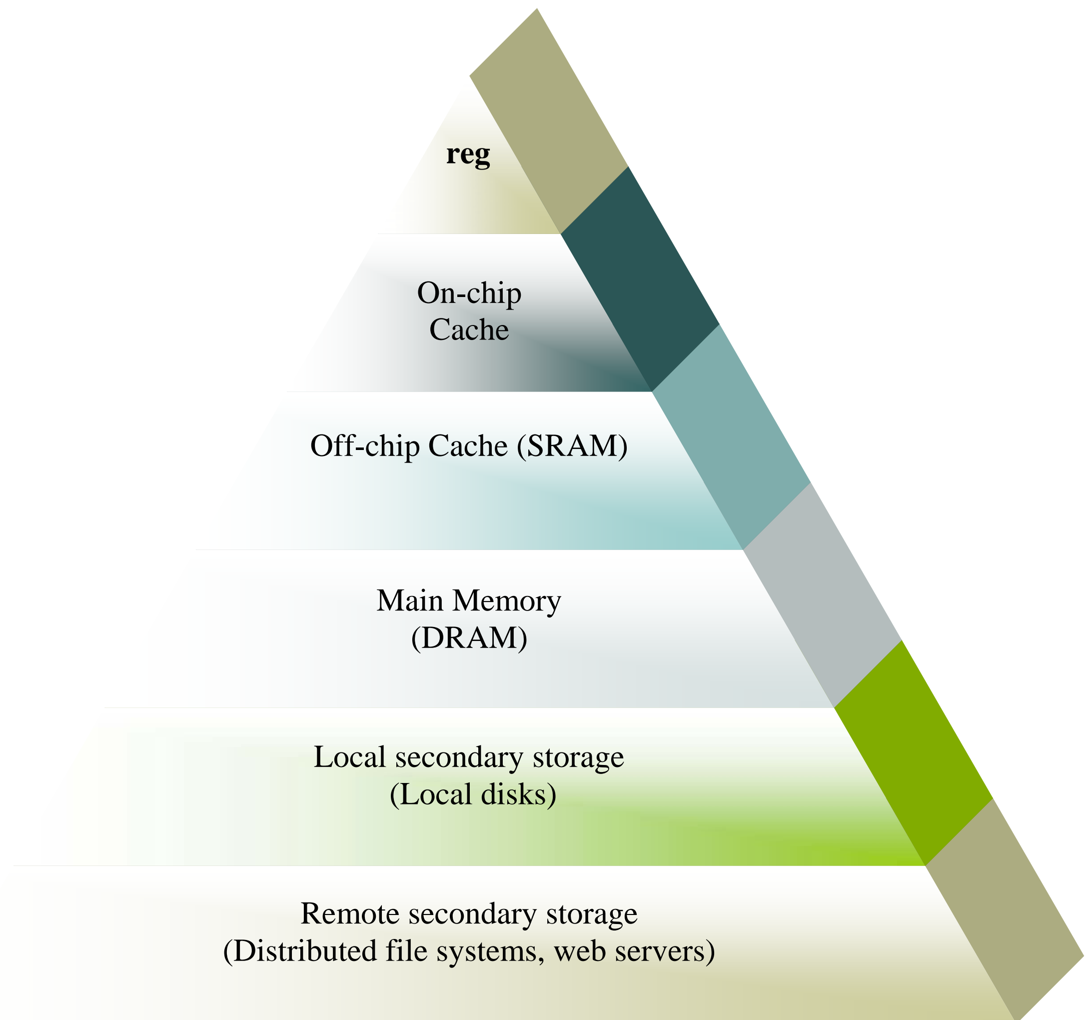

Figure 1.00: *Levels showing the abstraction involved in the memory hierarchy conceptual framework*

If you must really know some of the technologies used in making of storage devices, there is quite a lot!
Transistors are commonly used. This because they are easy to fabricate (based on silicon tech.) and are considerably cheap.
When arranged in particular ways, they can be used to store some limited amount of data. In tech talk transistors are referred to as **GATES**[1]. Gates are important to the operation of the **CPU**.
These gates operate using Boolean logic and are arranged in a way so as to be able to capture a bit and store it. (Storage is as either a 1 or 0, you should be fully aware the conversions from analog to digital that occurs, the high voltage represent 1 and the low voltage representing 0).

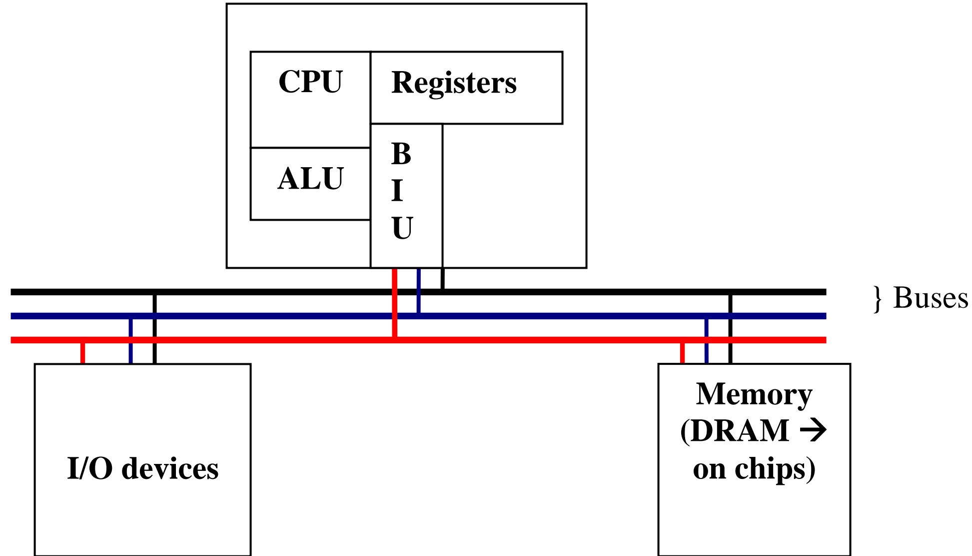

Figure 1.02: *the* ***von Neumann model****; the colored lines show the system bus. The system bus is divided in three: address, control and data bus. For purposes of illustration, let black be the address bus, red the data bus and blue the control bus. Address buses help the computer identify a location of the data; the control bus handles signals that deal with the logical flow of instructions such as how a bus transaction will occur; the data bus is used to transmit data either to be read, written, deleted or copied to another predetermined location.*[2] ***BIU*** *is the bus interface unit.*

---

[1]Gates are electronic devices, transistors to get straight to the point. Transistors come in all shapes and sizes (of course at a micro-scale). A transistor's mode of operation is very interesting. Charges are used in almost the most bizarre ways. Types of transistors: **FET** (field effect transistors), **MOSFET** (metal oxide semiconductor FET), **CMOS** (complementary metal-oxide semiconductor), **NMOS** (non-metal oxide semiconductor) and others. We shall discuss transistors and Boolean algebra (the basis of logic gate design) in greater detail later.

[2]Do some more reading on the von Neumann architecture

Data Structures

This refers to a structure (not really physical!) used to store data effectively so as to enable a reliable retrieval of the data in the future.
Well, in this course structure I proposed to you, we shan't go so deep into data structures, but trust me on this one, you will meet them quite soon!
Heading back to our previous topic, *Introduction to Python and computer programming*, I will just briefly talk about data types encountered in a programming language.

A programming language has a few characteristics. It ought to have a *syntax*, a set of rules by which programmers shall abide. The English language has a set of rules: capitalizing, putting a period (.) in the right place, typographic rules etc. The *syntactic rules* are what the programmer will use in his code (sorry about my gender insensitive approach; "his").
A program should be *semantic*, which means that it should have meaning both to the programmer and the computer (which will understand any *syntactically* correct program). By "meaning", a programmer should be able to infer all the logic involved in a computational process (or procedure).

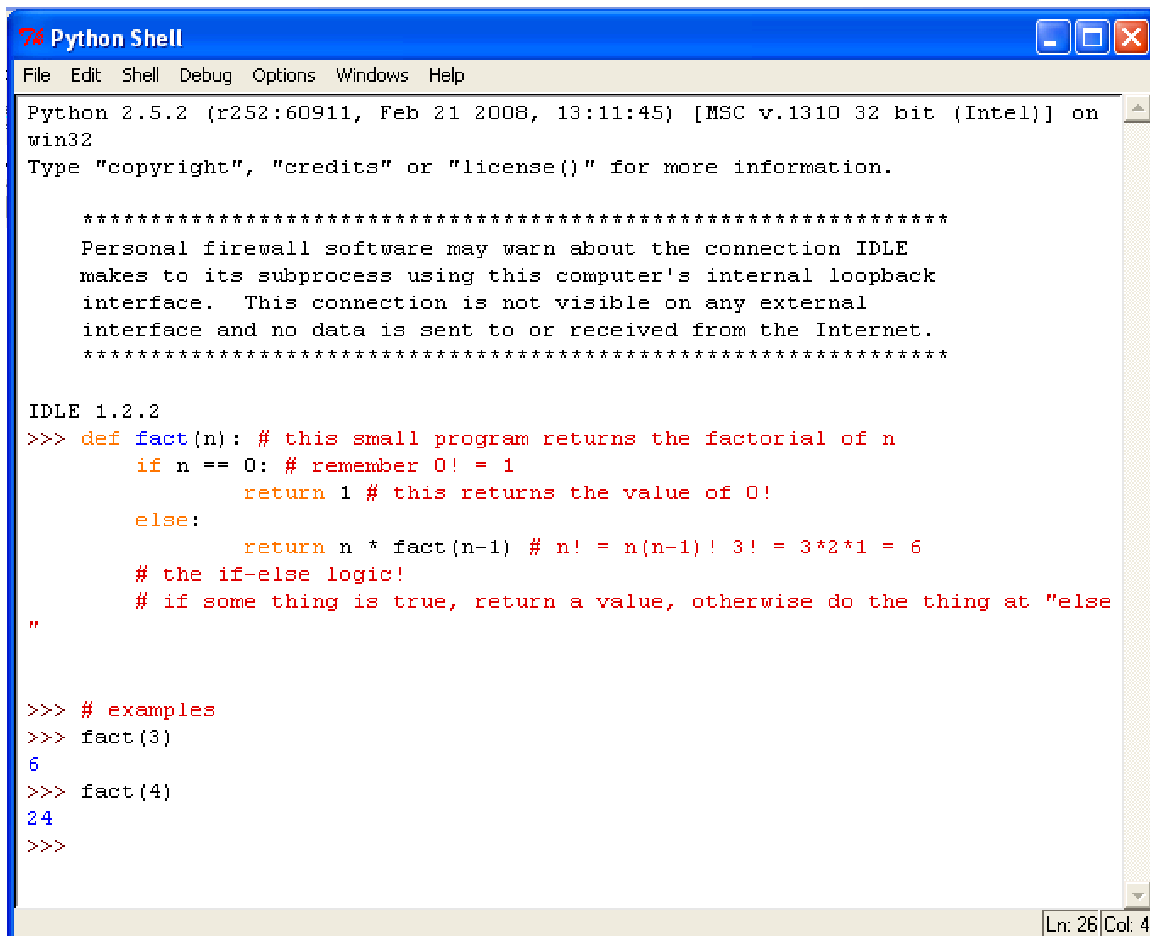

Figure 1.03: *Python's IDLE showing the factorial program*

Figure 1.03 shows a working model of the factorial program. It has got meaning because the instructions are clear and will produce an effective program. Another name of a program (also known as a function!) is an algorithm.
The factorial algorithm gives a sequence of steps to be taken in order to get factorial of a number **n**. This function/program/algorithm states that if **n** is not zero, then multiply **n** by a function with its argument reduced by 1 (**n-1**), if **n-1** is still not zero, continue doing that (multiplying the argument outside the function and reducing the function's argument by 1).
In effect, if we are to solve 3! think of the steps that our program shall follow are like this:

```
>>> fact(3)
>>> 3*fact(3-1)            # still function's argument (3-1) is not zero, take it out!
>>> 3*(2*fact(2-1))        # is the function's argument 0? No!
>>> 3*(2*(1*fact(1-1)))    # is argument zero? Yes
```

Eventually this looks like;

```
>>> 3*2*1*fact(0)
>>> 3*2*1*1
>>> 6
```

This process is also referred to as a recursive procedure; the function calls on itself several times!

In our **fact** program I followed *syntactic rules*. Some of Python's *syntax* includes not using keywords for function names, knowing when to use an integer or decimal.
List of some keywords used in python:

and	def	if	not	return	exec
assert	del	finally	import	or	try
break	elif	for	in	pass	while
class	else	from	is	print	yield
continue	except	global	lambda	raise	

Keywords are inbuilt commands that are specific to a programming language. Programming languages interface with other languages down to machine language. There are certain pointers that will be helpful when writing down programs; for example I used `def` to define my program `fact`. You will get a feel of more of these keywords when you start using the text I referred to you and hands-on programming exercises. These lecture notes are just an overview; I expect you to get baffled and excited by these so as to be

compelled to read even more and not expect to get everything from the teacher. I will help with the extremely difficult!

Types of data

In computing and mathematics we encounter various kinds of data. This data could be numbers and numbers could further be classified into **integers**, **floating-points** and **long-integers.** I don't mean to rule out the mathematical numbers in their entirety (e.g. $\Re$, real numbers, complex numbers, rational numbers, etc)
Other types of data we are faced with are **strings, tupples** and **lists**. I leave this to you.

One data-type I will talk about is the floating point. The floating-point is just another way of calling a number that has got a decimal point in it. Examples of some floating points are: 0.0, 1.0, 1.23, 34.534, etc.

Strings are mutable forms of data. In Python, they are represented as shown below:

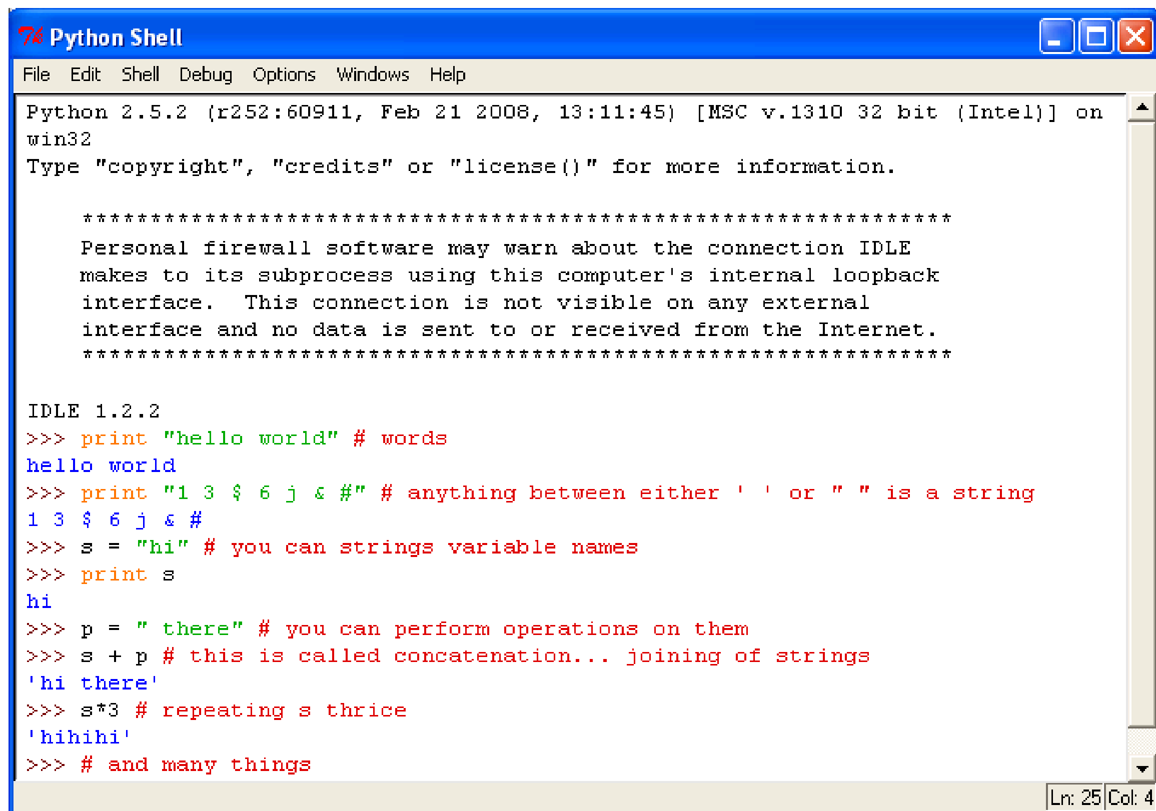

Figure 1.04: *Strings in action (explanations in red)*

If you find the Python book I sent is not enough, you can use the resources below to search for more info.
Some great sites to start with are http://www.wikipedia.org, http://www.scirus.com, http://scholar.google.com, http://google.com.

Research is good for you!

In our Python text are many examples of programs you will meet. To learn a programming language effectively, you ought to write as many programs as you can. You will get a rich experience out of excessive writing. I will be giving hints and advice on many issues pertaining to program design and software development.
You may be wondering why we don't start with Graphical user interface design in this first lecture. The reason is, we start small before going on to an advanced feature which we'll meet with object oriented programming languages and component based programming.
A strong background in number theory, mathematical logic, complexity and algorithms is advised. I shall give a less rigorous introduction to all that.
We shall first get down to basic algorithms while at the same time learning the features associated with the Python programming: the basic construct of any programming language.

The text will be very useful when used in combination with online resources (books), my lecture notes and practice.

These lectures will not dwell so much on Python, the assignments I will be giving you will do that.
It is from your own reading that you will learn a lot about the Python syntax and coding in Python. The series of lecture notes are mostly about technique that can be applied to all kinds of programming. Programming in other languages is very similar, the major difference they have is the syntax (just like a natural language like French or English; they have a grammar that is unique or nearly close to each other).

There are certain myths moving around the circles of Tech guys that certain languages are stronger than others. In my opinion, all languages are equal but some are more equal than others.
But, my experience with these languages is that, it is not the Language's name that matters (or does the trick), it is the way you use the program; hence Python™ will help you build your algorithmic design skill set. It is the simplest language I have ever encountered.
The syntax is left as an exercise to you. Feel free to ask if you have any questions.

Algorithms

An algorithm is a sequence of instructions that perform a computational operation. Computation operations can be arithmetic, others could involve the movement of files from one partition of a hard drive to another, an operation could involve identifying the shortest route to a destination (shortest path first algorithm **see OSPF**), and encrypting data files using cryptographic techniques.

An algorithm is a series of computational steps that transform a certain input(s) into their corresponding output(s)

My approach to teaching software engineering introduces algorithms at a much earlier stage in the course because I feel that a skilled programmer should be able to lay down his program on paper, test it on a computer system (involves documentation and debugging) and finally implement a feasible structure of the working program. This technique is known as the Testing-Before-Coding technique.
The algorithms I will be teaching are still at a basic level, the more complex ones will come as you grasp both the mathematical logic and number theory.
Number theory s largely concerned with properties of the set of positive integers such as primes, other theorems involving integers: Fermat's last theorem; $x^n + y^n = z^n$ for some $n \geq 3$ which deals with what is known as a *Pythagorean triple (x, y, z) from the sides of a triangle. In this case we are wonder whether that holds true for powers higher than two?* You will be surprised by the number of software applications in the computer world that are number theoretic; online transactions, communications systems onboard a plane, etc.)

I won't try to scare you much, only just to tell you to have a lot of fun…

We will do a slight advanced algorithm lecture when we meet physically, and shall therefore cover more advanced methods in algorithm design: merge sort, bubble sort, insertion sort, path searches (nice application in **AI**), heuristic search, uninformed searches, cryptographic algorithm design etc.

For now, the basic algorithmic design…

A computer as you now know is "STUPID"; garbage in, garbage out is the best term for it. What computer designers do is to just make it look intelligent, of course with some clever programming, you can make a nut case like the computer seem to be quite clever.

You will have to tell the computer how to perform computations…
Hence, the phrase; "programming a …"
You won't beg it to do you something it can't handle… a computer is a stupid quick learner!

***OSPF** is the Open shortest path first algorithm. It is used in data network communications to enable routers (internetworking devices) communicate with each other

“Kinds” of programming
Most programming is on the side of imperativeness… the HOW question, is of importance. Take a look at the mathematical expression below:

$$\mathbf{a} = \sqrt{\mathbf{b}}$$

When a computer is told to find the square-root of a number, it will follow a process.

What kind of processes do think can be used?

Let me give you just one hint:

- Newton-Raphson method

A human being would manipulate the expression above and get the answer, a computer on the other hand would use an algorithm.
The Newton-Raphson algorithm is really simple, and is covered under numerical methods in applied mathematics.

Newton-Raphson method

Suppose we have a certain variable $\mathbf{X}_n$ and it is the square root of a number, constant, C. (*You can use references if you wish).*

$$\mathbf{X}_n = \sqrt{C}$$
$$\mathbf{X}_n^{\mathbf{2}} = C$$
$$f(X_n) = \mathbf{X}_n^{\mathbf{2}} - C$$

We now have a function in x which we manipulated…
For this method to be accurate enough, we have to perform iterations (repetitions) until we get to a value that has the least error, as we shall see momentarily!

The method itself

$$\mathbf{X}_{n+1} = \mathbf{X}_n - f(\mathbf{X}_n) / f'(\mathbf{X}_n)$$

The above mathematical statement summarizes it all. But it can be understood well by illustrating with an example; finding the square root of 4.
The idea behind the algorithm:
Start with a guess, we know as $\mathbf{X}_n$, we shall call it $\mathbf{X_0}$, and later give it a value, say 1.5.
To give you a verbal rep of the expression: “the square root is $\mathbf{X}_{n+1}$ given that the ratio of the function and its derivative is subtracted from $\mathbf{X}_n$.
Newton realized that the ratio of the function and its tangent (derivative; dy/dx) is a close approximate to the value of its n-th root. When you read a large calculus text, you could get a deeper coverage of the method. I will just show its application.

Setting $X_0 = 1.5$,

$$X_1 = X_0 - (X_0^2 - C)/2X_0$$

$$X_1 = \frac{1}{2}\left(\mathbf{X}_0 + \frac{\mathbf{C}}{\mathbf{X}_0}\right)$$

$$X_1 = \frac{1}{2}\left(1.5 + \frac{4}{1.5}\right)$$

$$= 2.0833$$

Checking for the error…

$|X_1 - X_0| = 0.5832$

Yet, error in most cases is less than or equal to $\pm 0.5 \times 10^{-n}$, where n is the number of decimal places.

We still don't satisfy that, we repeat the above procedure, iterate, this time we shall be finding X_2 from our known X_1 value.

Remember that Error = $|X_{n+1} - X_n|$ (I used to have problems with mathematical notation, so I will clarify what the n's stand for in the expressions. n is just the nth value of X. If n=1, then that is the first value, 2, then the second, 3, then the third, and so on.

Computer scientists usually follow a different counting scheme known as the zero-based counting; we count starting from zero. So X_0 is the zero-th value.

In the above algorithm, it is kind of hard to get a straight 2, because of the truncations or rounding-off of figures that occurs. If you handle large decimals, you might get a straight two. Otherwise, it might be 2.000000003, or 2.000000013, or anything that is very close to 2, but not greater than two.

As an exercise, write down an expression for find the cube root, fourth root and fifth root of a number **[$\sqrt[3]{C}, \sqrt[4]{C}, and \sqrt[5]{C}$].**

[hint: rewrite the above Cs as $C^{1/3}$, $C^{1/4}$, and $C^{1/5}$]

Variables and more of the Python™ syntax

Variables are important to programming. And are therefore given names and assigned to values (either Boolean or other). A variable, therefore, refers to a value; it is also the place in the computer's memory for storing a value.

An assignment statement is used to give a variable a value… (use "=" symbol when assigning)

These assigned values have certain types (**int, float, str**) which you've already seen.

```
>>>a = "hello, world"
>>>print a
>>>Hello, world

>>>def sum_of_two(a,b):
        return a+b
>>>#when the above function is tested on 3 and 2, we should get 5.
>>> c = 3
>>> d = 2
>>>sum_of_two(c,d)
>>>5
```

Variable names

When giving variables names there are certain syntactic rules to follow.

- Never use the keywords of the programs
  e.g. def #this is used to define a function (or program), and, if, else, etc. Python™ has about 29 keywords; these are reserved for special use
- Other rules are not to use certain punctuation marks ("*", ".", ",", "!", etc) for variable names
- Do not use many symbols: #,$,%,^,&,*,@
- Give logical names, names that will help you think as you write you code

Check out the Wiki book on Python™. I hope you keep the books I send in a special folder; you can even encrypt it… password protect it, and hide it in the school's system. We wouldn't want any malicious stuff happening to your books! (Just right click the folder, click properties, check the hidden box; a tick, go to the advanced tab and click encrypt)

If the concept of variable names is baffling, just think of them as mere representations of the input your algorithm (function machine) needs to produce an output (from our definition of an algorithm)

There is no big difference between a program, software, and a function; a program is just a collection of functions strung together so as to work with each other to produce a desired result.

Style

There is no programming language, no matter how structured, that will prevent programmers from writing bad programs (L. Flon).

It is the nobility of their style which will make our writers of ***1840*** *unreadable forty years from now (Stendhal).*

Style is about good coding practices. When developing a program today, and not inserting some form of documentation will lead us total mayhem on the day we discover certain bugs in our programs.
It is with high regard that I write this; always leave bread crumbs so you can back-track your footsteps in the long journey of coding.
Style is "lethal"; it separates the good tomatoes from the bad ones. Coding with style will ensure that your program is very readable… has got an introduction, a body and a conclusion… talk about some high school English here!

In the life-cycle of a program, programmers spend most of the time maintaining, upgrading, and debugging existing programs. So it is in your best interest that you leave those bread-crumbs.
In Python™, commenting is done using #. And it will work for just one line

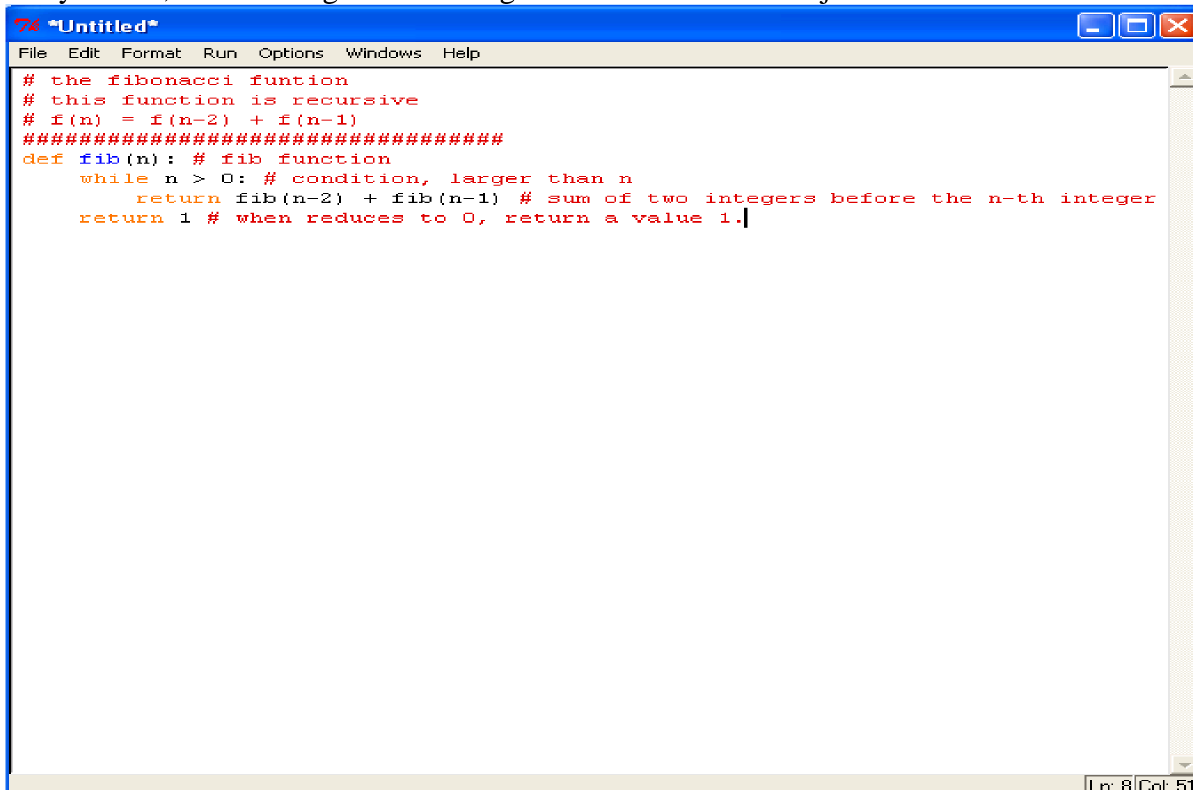

Fig 1.05 *illustration of commenting with the Fibonacci function*

Mathematical trivia

The Fibonacci sequence is created from the mathematical function $f_{\mathbf{n}}$ shown below:

$$\{f_n\}_{\mathbf{n=0}}^{\infty} = 1, 1, 2, 3, 5, \ldots$$

$$f_{\mathbf{n}} = f_{\mathbf{n}-2} + f_{\mathbf{n}-1} \quad \forall\, \mathbf{n} \geq 2 \qquad \mathbf{n} \in Z^{+}$$

{The symbol $\forall$ denotes the phrase; “for all values of” n, in this case.}

This program uses a **while** loop to carry to mimic the process described in by the function above. We start counting the elements in the sequence from 0 (**remember:** zero-based counting). So the zero-th number is 1, the first is 1, the second is 2, the third is 3, the fourth is 5, and so on. For numbers greater-than-or-equal to 2, they are just the sum of the previous two integers in the sequence.

n	n-1	n-2	f_n
0	#nul	#nul	1
1	#nul	#nul	1
2	1	0	2
3	2	1	3
4	3	2	5
5	4	3	8
6	5	4	13
7	6	5	21
⋮	⋮	⋮	⋮

More on commenting

Other languages like C and Java use /* <comment> */ for commenting, where <comment> refers to whatever you write as your comment.

At the beginning of a program is a comment box with information about the program. You can box your comments to make them stand out i.e.

```
##############################
####### Fibonacci function ######
##########Author###############
#### Alvin Roy ################
##########Date##################
######06/07/08##################
#/*********************************************************/#
#        this can grab your attention                    #
#                                                        #
#/******************************************************#
```

[**note:** Python's interpreter will not read anything after the # symbol]

Not every program will need this kind of commenting, so you can apply what is really necessary.

To work on your comment skill you can reverse engineer someone else's code, and see what they write for their comments.

A typical format for program commenting:

- Heading; containing the name of the program or function is there are many
- Author; you need to give yourself some credit especially after hustling with all that tedious coding to make the program functional
- Purpose; why did you write the program? You could tell the audience ☺
- Usage; a short description of how the program runs

Oualline's law of Documentation:

"90% of the time the documentation is lost. Of the remaining 10%, 9% of the time the revision of the document is different from the revision of the document and therefore completely useless. The 1% of the time you actually have the documentation and the correct revision of the document, the document will be written in Chinese."

How to avoid Oualline's law from taking its toll on your technique and sweat

To avoid this law from having its impact, always put the documentation in the program itself. Documentation is at times thought of as the manual! ☺ It is just the commenting.

- *Referencing*; when you use someone's code, and modify, it is wise to give the guy his due credit otherwise it just becomes copy right violation. So give credit where it is due.
- *File formats*; soon you will cover programming that deals with object handling, file handling… so you will be required to give a few comments on the file formats you have used. Whether you used a bit-mapped image (*.bmp), a JPEG, an MP3, a *.txt, *.dat, *dll, etc.☺ All such file extensions [*.<blah>] are important. In a large working program, these are all used
- *Restrictions*; you should list the limits that your programs handles. Whether it just works on integers alone, floats alone, strings (for the case of data types) or a particular file format. These are usually just the error messages you design to give the user feedback of his/her input e.g. "Error. No integers", "The program doesn't check for input errors", etc.
- *Revision history*; this refers to the chronology of events that have occurred since the program/software was released. It gives credit to the guys who've worked, modified or peeked through your program.
- *Error handling*; you should write what the program will do if it detects an error. Whether it will inform the user to redo something or it will just terminate
- *Notes*; include special comments or other comments that you may have not placed in the program before

A note on indentation

Indentation helps the program organize his/her thoughts. I expect that you have read the first four chapters of the book by now.

Good programming technique

With experience and practice, you will develop your own techniques in writing powerful code!
Practice! Practice! Practice

Assignment 1

Do all the assignments at the end of chapters 1, 2, and 3.

Problem 1

[Hint: read chapter 4, and the wiki book on Python™, the Python reference manual on the topic of raw.input() and input modules… work should be done in script mode and not interactive mode. Python works in two modes.

The interactive mode is denoted in Python's **IDE** by >>> and the script mode is just like any basic text editor like notepad. This problem will test your recursion, iterations and how to get stuff from a person. Most of the earlier programming training doesn't expect the novice program to work with data that is input into the system. You are the exception; and you have about $2\frac{1}{2}$ months to do this work.]

Write a program that does the following in order:

1. Asks the user to enter his last name.
2. Asks the user to enter his first name
3. Prints the user's first name
4. Prints the user's last name

The output of the program (with sample user inputs) should look like this:

```
Enter your last name:  Roy
Enter your first name: Alvin
Alvin
Roy
```

1. Save your work as ps1.py
2. Submit your work via e-mail, with your explanations written out in comment form
3. You have 2.5 months

Format of work/work layout

```
# Problem Set 0
# Name: Alvin
# Time: 1:30
#

… your code goes right here
```

Problem 2

The following program has two mistakes in it that cause it to print an incorrect answer

```
# Computes how much interest is earned on a give amount of principal.
Principal        = float (raw_input (‘Enter Principal: ‘))
interestRate  = float (raw_input (“Enter interest rate as a percent: “))
years             = int (raw_input (‘Enter number of years: ‘))
totInterest     = 0.0
curYear         = 0
while curYear <= years:
        annualInterest = principal * interestRate
        totInterest = totInterest + annualInterest
        principal = principal + annualInterest
        curYear = curYear + 1
print ‘Total interest in ‘ +str(years) + ‘  year(s): $’ + str(totInterest)
```

Here is an example of a test case which yields an incorrect answer (I implore you to get more domain knowledge on the subject of finances, annuities, interest computations)

```
Enter principal: 1000
Enter interest rate as a percent: 10
Enter number of years: 1
Total interest in 1 year(s): $120000.0
```

[**Hint:** get rid of the **while** statement]

The correct answer is $100 and not $120000.0

Fix the above program, and use the test just given to verify correction of the program. Save the work as ps2.py