

PLD Text (4th Edition) Chapter 1 Notes

Chapter Learning Objectives

- Computer components and operations
- Simple program logic
- Evolution of programming models
- Steps involved in the programming process
- Pseudocode and flowcharts
- Program comments
- Programming and user environments

Chapter Notes

These chapter notes are my lecture notes and they're provided for you to use as a study guide and supplemental resource. Please keep in mind these are *not* a substitute for the textbook and reading the chapter. The goal is to point out some of the key concepts in the chapter and to explain/clarify "difficult" aspects of the chapter material. Additionally, I may also provide some code examples that you may use for class assignments.

The textbook used in this class is "programming language neutral" and emphasizes various concepts that apply to virtually all programming languages. The thought is that if you understand the concepts that apply to all programming languages in general, it should be much easier for you to learn a particular programming language.

We will be using the Java programming language to help reinforce the various chapter concepts so I will be adding in Java programming material as it relates to the chapter material and the programming assignments in this class. Learning Java now will provide you a good foundation for a regular Java programming class as well as languages such as C++ and C#.

Key points:

- Hardware consists of the physical devices – motherboard, video card, disk drives, etc.
- Software is what makes the hardware actually do something.

Three fundamental computer operations:

1. Input – from input devices such as the keyboard (user data entry), storage (i.e. file or database, scanner (images, documents), etc. In essence, input devices provide ways for raw data to enter the system.
2. Processing – raw data is transformed in some manner to provide “meaningful information”. This includes checking for accuracy, calculating totals or calculating other desired results.
3. Output – “transformed” data is displayed (*display* typically refers to some sort of video display) or printed out in some format that provides meaningful information to the user.

Related to input and output:

- Storage – both raw data and processed information is typically stored in memory or some sort of a file (external storage) to allow retrieval for future use. When we talk about storage, computers have internal storage often referred to as RAM (Random Access Memory), *main memory* or *primary memory*. In any case, it is used for storing data being immediately processed or otherwise used by the program as well as the actual program instructions. This is *volatile* memory meaning data is lost once power to the system is lost or the program terminates. *Secondary* or *external storage* is non-volatile, permanent storage outside of main memory but not necessarily outside the computer system “box”. This includes storage devices such as the hard disk, magnetic tape, CD-ROM and all those nifty USB storage devices.

Other points from this section:

- Programs or applications consist of various instructions and are written using a programming language. Numerous languages exist.
- Each language has a certain syntax or grammar that must be followed. This is covered in more detail in the next section.

Understanding Simple Program Logic (Pages 4 – 6)

Programs or applications or “apps” are written using a programming language. Learning a programming language is much like learning Spanish, French, German or some other foreign language. Like learning any language, it takes time, practice and *lots* of patience!

Every language in existence has a certain *syntax* or *grammar* that defines a valid sentence. Program languages are no different. Each one also has a certain syntax or grammar that defines a “valid instruction” or *program statement*. We’ve all learned English grammar in various English classes we’ve taken in school over the years. If somebody speaks or writes to us using poor grammar, we, being the smart humans we are, can usually figure out (eventually) what that person is trying to say to us. We’re not so lucky with computers! Each instruction must be perfectly coded or we get syntax errors. By using the NetBeans Integrated Development Environment (IDE), the code editor will tell us when we have syntax errors so we can fix them before we try running the program.

Page 4 (middle) - Besides getting the syntax right, the textbook illustrates the need to get the program logic right:



This example has correct syntax but contains *logical* or *semantic* errors. The steps are out of order, one of the steps makes no sense for the task we’re trying to accomplish, and we would very likely burn down the house actually trying to do this! When you see those little “Don’t Do It” callouts throughout the book, pay close attention and take them seriously as things to avoid doing.

Speaking of syntax, a programming language that may be English-like and easy for us to read must be translated into machine language (1’s and 0’s) by either a *compiler* or *interpreter* before it can actually be executed by the computer. A compiler breaks the instructions down into machine language and produces an executable file. If you poke around on your Microsoft Windows PC, you’ll find that executable files usually end with a .EXE or .DLL file extension. An interpreter translates the high level instructions on-the-fly as the program executes. Java is considered to be an interpreted language. However, Java code is broken down (compiled) into *byte code* before it is executed by the Java Runtime Environment.

If you look at the sample program shown on page 5, you’ll notice it uses English-like statements. These statements are often referred to as pseudocode. It does illustrate the three fundamental steps of input process and output.

```
input myNumber
myAnswer = myNumber * 2
output myAnswer
```

One trick to help get the logic right is to keep in mind that every program you will ever write will consist of input, process and output. Consider the number doubling process on page 5. The textbook explains the process very well.

Two different types of software:

- *System software* - typically comprises the operating system (i.e., Windows, Linux, Mac OS, etc.) that allows us to make use of the computer hardware such as the processor, memory, storage devices, video display, keyboard, mouse, printer, etc.

- Application software allows us to accomplish various tasks such as process payroll, surf the Internet, create documents, manipulate images, etc.

In both these cases, note that a program is a set of instructions that accomplishes a particular task.

Quick Programming Language History

This isn't in the text but here's a quick overview of some of the major languages that have emerged over the years.

The earliest, first generation languages (from the early 1950's to early 1960's) used binary machine instructions. Instructions were typically entered by setting toggle switches on the machines.

Second generation assembly languages started emerging about the mid 1950's. Assembly languages used *mnemonic* (abbreviated but supposedly easy to remember) instructions such as ADD for add, SUB for subtract, STI for Store Immediate, JMP for jump, etc.

Third generation (high level) languages began emerging in the 1960's. Instructions for these languages were more English-like and allowed programmers to focus more on the program's business logic rather than low level details such as manipulating registers on a processor and memory management. Additionally, many of these languages provided "libraries" of pre-written code for various programming tasks. Languages such as COBOL (Common Business Oriented Language) were optimized for developing business applications while languages such as FORTRAN (FORMula TRANslator) were optimized for scientific/engineering applications.

Fourth-generation languages (4GL) developed in the 1980s were closer to human languages. While much easier to learn, 4GLs are generally slower to compile. Early proponents of 4GLs thought that 4GLs would allow non-programmers to create their own programs. 4GLs were marketed at that time as eliminating the need for programmers! Although there are now many tools that allow end-users to query and manage their own data, programmers, analysts, and system designers are still very much in demand for their specialized knowledge and their ability to understand and create large, complex application systems.

Fifth generation languages developed in the 1990s like PROLOG are used primarily for artificial intelligence, fuzzy logic, and neural network applications.

The C++ language emerged about 1983 and, while there were some object oriented languages that emerged well before this time, C++ provided a migration path from the highly popular C programming language to object oriented programming where programmers could learn object oriented programming in a familiar environment.

Java emerged around 1995 and its original objective was to let devices, peripherals and appliances possess a common programming interface. It incorporated syntax similar to C++ to simplify the learning curve but provided some improvements such as *garbage collection* to manage the release of unused objects from memory.

Evolution of Programming Models (Pages 6 – 8)

Over the years, a number of programming languages have emerged. The programming language history in this outline provides a good illustration of that. The newer languages have become more English-like, support *variables* (giving meaningful names to memory locations), and coding *functions*, *procedures* or *methods* to break the program into more manageable and reusable modules. The newest languages support object oriented programming.

Two primary programming techniques (page 6):

- Procedural – Emphasis is on the tasks or procedures that manipulate the program data. A large, complex programming problem is broken down into numerous small tasks using what is often referred to as a *top-down design*. Data is

effectively passed from one procedure/task to the next. Various tasks are accomplished in a step-by-step manner. This is a substantial improvement over unstructured coding in the fact the program is logically broken down into modules (procedures and functions). Data is passed to the procedures and/or functions with each performing some defined task on the data.

- Object – Emphasis is on creating objects that interact with one another to accomplish required system tasks. This offers an additional improvement of procedural techniques by “packaging” data and the code that operates on it together in a well-defined box with an emphasis on reusable code that can be used in future projects. Additionally, we, as humans are accustomed to working with objects – our world is full of them! Object oriented design/programming still uses many procedural design techniques .

Both approaches work and the primary difference between the two is in planning the project. Object oriented programming does in fact incorporate procedural programming techniques. The text makes the comment, *“You can write procedural programs in OO languages, but you cannot write OO programs in procedural languages.”*

Object oriented programming techniques are extensively used for Graphical User Interface (GUI) programming. All those buttons, text boxes and other components you see in a “typical” Windows form are all objects. You will be working extensively with these – both as the programmer and as the end user over the course of the semester.

Software or application design, in a nutshell, is very simply a process of taking something that is complex and systematically decomposing or breaking it down into smaller pieces that can be more easily understood and translated to computer code. Given a large, complex application like a financial system, it's doubtful anybody would be able to immediately start writing code for it. However, if we break the system down into pieces or *program modules* such as accounts payable, accounts receivable, general ledger, etc., and then break those pieces down further as needed, things (eventually) become more understandable to a point where we can start writing code. An added benefit of breaking a system down into small pieces is that we can now allow multiple developers to work on the project.

A number of software design methodologies have also emerged over the years. All share the same common goal of taking a large, complex problem and decomposing it.

This section covers the 5 fundamental steps of the software development process. When we talk about software and computer programs, most modern textbooks refer to programs as *applications*. The two terms are synonymous though. Numerous software development methodologies exist but all of them incorporate the following steps in one form or another. My notes and numbering here differ slightly from the text.

1. Requirements (Analyzing the system/understanding the Problem) – System requirements are defined. At this stage, the key word is *what*. What is the system expected to do or accomplish? Key things are to look at the system through the end user's eyes. If I, as the user, am going to sit at the computer and use the application, what task(s) do I expect to be able to accomplish? The focus is completely on desired functionality.
2. Design (Envisioning the Objects) – At this stage, the key word is *how*. How are we going to implement the system? We define the objects we need and how they'll interact, determine how we're going to store/retrieve data, etc. At the design stage, we also design and plan the logic behind the "behavior(s) of each object (see page 9, "Designing the System" and page 10, "Developing Program Logic"). Also, an important part of planning the logic is a *desk check*.
3. Coding – At this point we are writing the actual program code in whatever programming language we have chosen for the project. Ideally, we should also be doing at least some basic testing of the objects as we code them. Part of initial testing includes compiling the program and correcting any syntax errors found. When we initially compile a program, one of the things the compiler does is provide us a list of *syntax errors*. Syntax errors are instructions the compiler does not recognize and are typically typographical errors. Before a compiler can do its job of breaking our code down into executable machine code, these errors must be corrected. Some of the newer code editors will now highlight lines of code containing syntax errors.
4. Testing – this phase involves doing a rigorous test of the program to ensure that all required functionality has been implemented and each feature produces the correct results. The process of locating and correcting errors or *bugs* in a program is referred to as *debugging*. Software testers run programs using a variety of input data. Some testers run only black box tests, in which they provide input and determine whether the output is valid without looking at how the program works internally. More complicated programs often also require white box tests, in which the tester looks at how the program works to make sure every possible logical path is tested.

Once tested, the application is normally released to the end users for daily use. It is often necessary to train users for large, complex applications.

5. The last phase is commonly referred as the maintenance phase. At this point, the program has been "put into production" and delivered to customers. New features are added, bugs not found in the initial coding & testing phases are corrected.

Speaking of software bugs, there are 3 different types:

1. Syntax – As mentioned before, these are typically typographical errors or otherwise miscoded instructions the compiler doesn't recognize and must be corrected before the program will compile and run.
2. Run time errors – Often referred to as *exceptions*, these occur when the program is running and are caused by things such as invalid mathematical computations (dividing by zero), trying to process a file that doesn't exist or is corrupted, connecting to a database with an invalid password, etc. Most modern programming languages (including Java) provide programming constructs to detect and allow a "graceful" recovery. We'll learn more about this in chapter 10.
3. Logical – The program compiles, runs, "goes through the motions" but doesn't quite deliver the expected results. Totals might be wrong; a computation result incorrect, data skipped that should have been processed, etc.

This is one of the most important sections in the book. Even if you never write another line of code again after leaving the class at the end of this term, being able to analyze and decompose a problem into smaller, more easily solvable pieces is a highly valuable and marketable skill!

Pseudocode and flowcharts are very commonly used for planning an application's logic. When we get to chapter 11, we'll also explore some additional design tools using Unified Modeling Language (UML) models.

Regardless of whether we're doing procedural or object oriented programming, when we design our application, we normally break the large programming problem into smaller, more workable pieces unless it's already a very simple program. Those "smaller pieces" consist of the various required actions or behaviors that are broken out into *functions* or *procedures*. In object oriented languages, the functions and procedures are often simply referred to as *methods*. For now, just think of a function/procedure/method as a set of statements or instructions that perform a specific task. We'll take a deeper look at functions and procedures in a later chapter and the difference between them.

When we talk about defining a program's logic, one term presented in this text that you'll often see in other textbooks as well is *algorithm*. An algorithm is very simply a step-by-step procedure to solve a problem or complete a task. Think of a recipe for cooking or baking something as an algorithm.

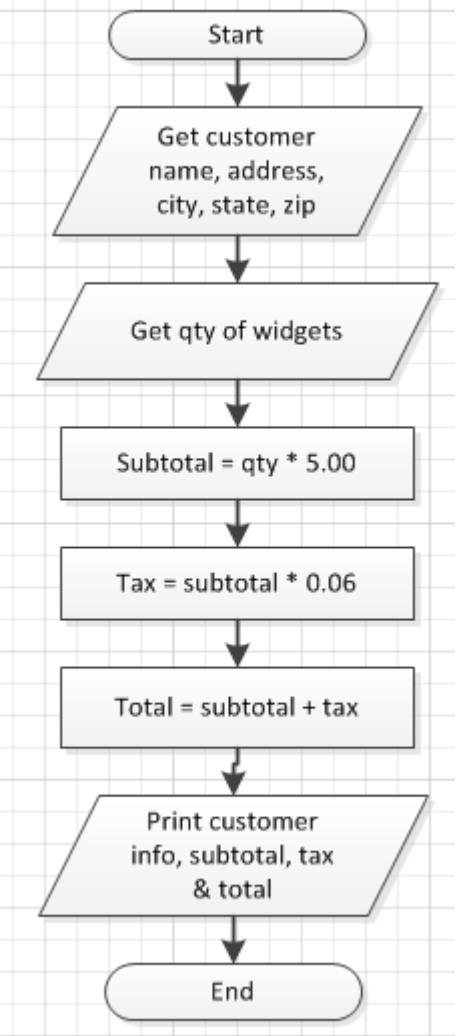
If we take the time to do define our program logic, writing the actual program code in the language we've chosen is always a much easier task so that's the motivation for doing this.

Both techniques presented in this section provide us a tool to define and document how the logic of a program (or sub-program) is supposed to work. Both techniques work equally well so, as for the question of, "which one do I use?", it's really a matter of personal preference.

Flowcharts – these have been around for many years and present a pictorial representation of the steps to solve a problem. There are numerous commercially available software tools available for creating flowcharts. The most basic symbols are shown on page 18. The oval, or *lozenge*, as the author likes to call it, shows the start and stop points. The rectangle shows some sort of process or computation. The parallelogram shows input (getting data) and output (displaying results, etc.). In a later chapter, we'll take a look at the *diamond* or decision symbol. The symbols are always connected with flow lines.


Pseudocode – English-like representation of the steps to solve a problem. There is no set way of writing pseudocode and this is very often what makes it difficult at first. Everybody has their own style so the trick is to come up with a style that works for you. It can be done easily on any word processor, text editor, even a spreadsheet!

To illustrate and compare both techniques, we'll do a side-by-side flowchart and pseudocode of the steps to solve a very simple order entry problem for Acme Widgets. They have tasked us with developing a program to get a customer's name, address, number of widgets and calculate the price, add sales tax and print a simple bill with the order information and total. Widgets are \$5.00 each and sales tax is 6%. Customers walk into the store to purchase so payment is made at the time of purchase. All sales are cash or check.

Flowchart (done with Microsoft Visio)	Pseudocode
 <pre>graph TD; Start([Start]) --> GetInfo[/Get customer name, address, city, state, zip/]; GetInfo --> GetQty[/Get qty of widgets/]; GetQty --> Subtotal[Subtotal = qty * 5.00]; Subtotal --> Tax[Tax = subtotal * 0.06]; Tax --> Total[Total = subtotal + tax]; Total --> Print[/Print customer info, subtotal, tax & total/]; Print --> End([End]);</pre>	<p>Pseudocode</p> <p>Begin</p> <p>Get customer name, address, city, state, zip</p> <p>Get Qty of widgets</p> <p>Subtotal = Qty * \$5.00</p> <p>Tax = Subtotal *0 .06</p> <p>Total = Subtotal + tax</p> <p>Print customer info., subtotal, tax and total</p>

When writing programs, it is very desirable to include comments to describe tricky sections of code or the function/purpose of a code module. In languages such as Java, C++, C and C#, anytime you see two forward slashes (//), anything after that is regarded as a comment. Java, C++, C and C# also support block comments as shown below. The “/*” is the beginning of the comment block and the “*/” is the end.

In Python, the pound or hash character (#) is used for adding comments to code.

 multiply.py - C:\Cuyamaca\2019FallCS119\Labs\Lab01\multiply.py (3.7.3)

File Edit Format Run Options Window Help

```
#!/user/bin/env python3
```

```
# Your Name Here!
```

```
# Cuyamaca College CS-119
```

```
# Lab 1, exercise 1, multiply 2 numbers
```

```
# declare variables
```

```
number1 = 0
```

```
number2 = 0
```

```
product = 0
```

```
# input
```

```
number1 = int(input("Enter first number: "))
```

```
number2 = int(input("Enter second number: "))
```

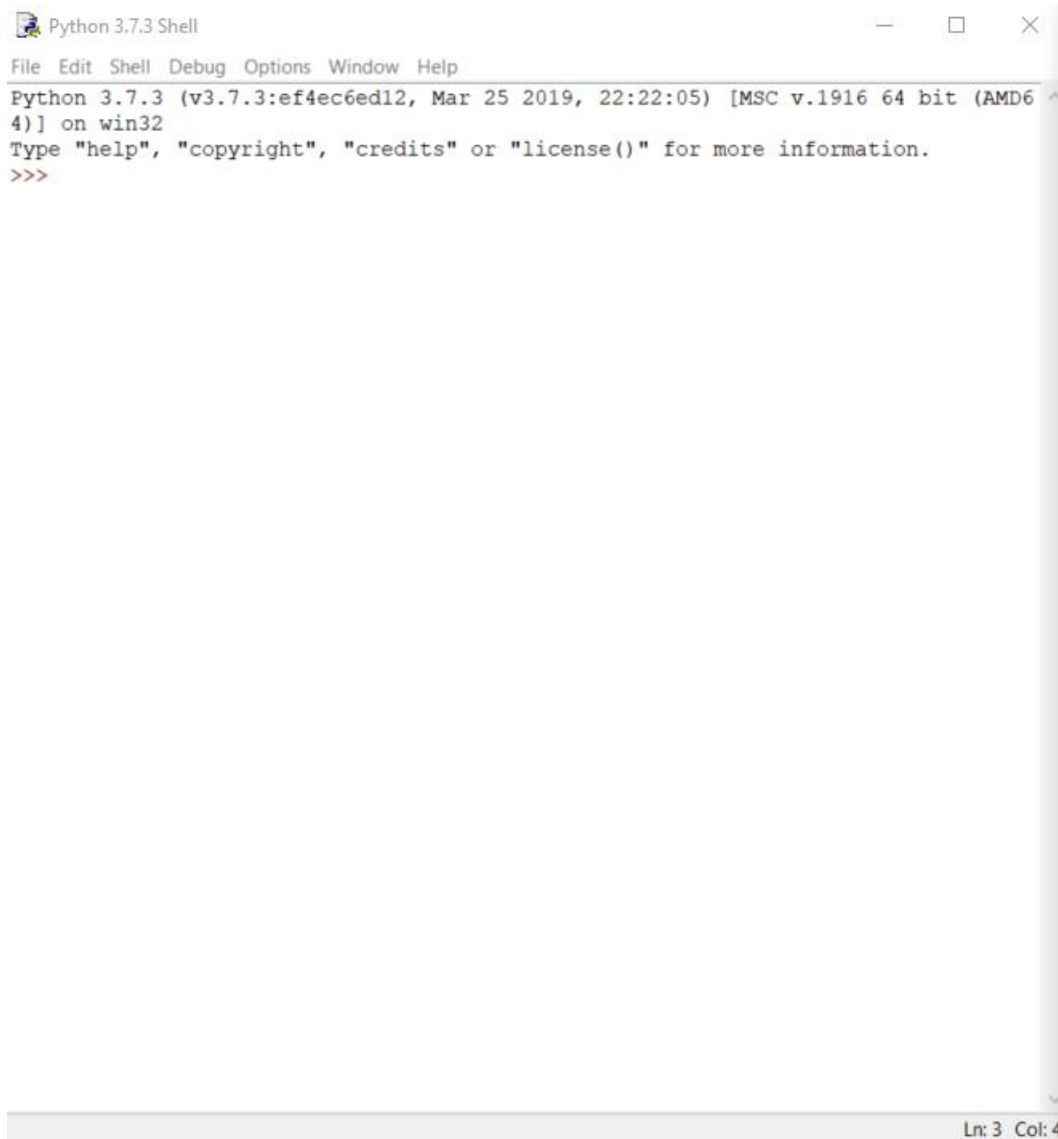
```
# process
```

```
product = number1 * number2
```

```
# output
```

```
print("The product is " + str(product))
```

Flowcharts and pseudocode can be written by hand but there are also a variety of applications we can use as well. When writing code, we can use a simple text editor with a command line compiler or use an Integrated Development Environment (IDE) such as Python IDLE with the code editor, compiler and debugging tools all nicely integrated together. It's really a matter of preference but, considering Python IDLE can be obtained at no cost, there is really no practical reason to *not* use such a development tool.



This section covers console (command line) and GUI user environments and is pretty self-explanatory.

The following supplemental sections provide some additional explanation and clarification of chapter concepts that tend to be confusing and difficult.

Analyzing and Solving Problems

One of the fundamental skills to be gained from this class is learning to analyze a problem and then develop a logical set of steps that will solve the problem. Even if you never write another line of code again after finishing this class, being able to effectively analyze and solve problems will put you ahead in any career you pursue.

Virtually all problems are initially too complex to solve on a computer coming right out of the gate. Many are too complex for even us as humans to fully understand at first. If we don't immediately understand the problem, that's okay but how do we "wrap our heads around it"?

One aspect of software design and problem solving not emphasized in the book but most definitely important in the real world is to involve end users, managers and other project *stakeholders* in the design and development process. Meet with the project stakeholders, ask lots of questions, observe the working environment, etc. See the system through the eyes of its users. Learn to spot and eliminate things that really have nothing to do with the system we're designing.

Software analysis and design is a game of *divide and conquer*. That is, we take a large, complex problem and systematically break it down into smaller pieces we can more easily understand. Very often, we take the smaller pieces and break them down into even smaller pieces yet and repeat the process until we (eventually) have something we can easily understand and code a solution. Once we've coded solutions to all the small problems, we ultimately end up with a solution to the initial "big problem".

A fundamental and very effective design strategy presented in the text is to take a problem or, perhaps, a smaller sub-problem and break it down into the steps of input, process and output.

For input, ask the question, "What input data is needed to solve this problem?" Be sure to identify all inputs needed.

For processing, ask the question, "What transformations are needed to the input data to produce the desired information?" Be sure to clearly identify any formulas or methods of calculation needed. This makes coding much easier later on. Keep in mind you may need to first make some conversions to the data (i.e. convert standard to metric units, etc.)

Once we've done our processing, the last step is output where we provide our result back to the user. How do we provide that result back to the user? Display it on a computer monitor? Print it? Write it to a file or database?

Here's a simple problem from page 29, exercise 5 in the text with a little added twist. Let's analyze the problem and break it down:

It's a bright, sunny day and Cuyamaca Learning Center has hired you as a consultant to develop an application that will accept two input numbers from the user and display the product of the two values.

In real world software design, we will often encounter things in the problem statement that really have nothing to do with the problem. First step: Eliminate those things! In this case, does the statement that it's a nice day and we've been hired as consultants really have anything to do with the problem? No! Let's get rid of it!

Also, clean up any inconsistent wording. In this case, is *numbers* and *values* the same thing or two different things? When in doubt, ask the customer!

~~It's a bright, sunny day and Cuyamaca Learning Center has hired you as a consultant to develop an application that will accept two input numbers from the user and display the product of the two values numbers.~~

We all agree the above version of the problem with all the nonsense removed and the inconsistent wording corrected already makes things a whole lot easier, right?

Next step: Let's break it down into input, process and output.

What input do we need? We need 2 numbers

What is our process? We need to multiply the 2 numbers. Our formula is: $\text{product} = \text{number1} * \text{number2}$

What is our output: We need to display the result

Guidelines to Writing Pseudocode

A key thing to keep in mind about the textbook is that all the “code” examples throughout the book are *pseudocode*. Pseudocode literally means false or fake code. Do *not* mistake the examples in the textbook for Java or any other programming language code!

Why bother writing pseudocode if it’s something completely fake? Pseudocode is a *design tool* that helps us organize our logic and our approach to solving the problem at hand. Another way to look at the whole thing: In other classes such as English, History, etc., we’ve all written that dreaded research or term paper. Before writing the actual paper, we write an outline to organize our thoughts, ideas, facts and the way we wish to present them. A well written outline saves us time and work writing, editing and revising the actual paper. In programming, pseudocode serves the exact same purpose. Well written pseudocode substantially reduces time and effort coding the actual application.

One thing that makes pseudocode quite confusing and difficult at first is that there is no set way to write it – everybody develops their own style. We don’t need to worry about conforming to the exact syntax of a particular programming language. **The key here is to develop a style that allows *you* to understand the problem at hand and be able to develop a solution.**

In the last section, we analyzed a problem and clearly identified the inputs, processing and outputs needed for a solution. It’s copied here for easy reference:

~~*It’s a bright, sunny day and Cuyamaca Learning Center has hired you as a consultant to develop an application that will accept two input numbers from the user and display the product of the two numbers.*~~

What input do we need? We need 2 numbers

What is our process? We need to multiply the 2 numbers. Our formula is: $\text{product} = \text{number1} * \text{number2}$

What is our output: We need to display the product

Let’s write some pseudocode for the above problem.

For input, we typically use the words *get*, *input*, or *enter*. Specific wording is a matter of personal preference. We’ll keep it quick and simple here and use *get*. It helps to use the variable names you intend to use in the actual application.

```
// input
Get number1
Get number2
```

For processing, pseudocode all formulas identified during analysis. Note the variable on the left side of the equal (=) sign contains the result of the calculation.

```
// process
product = number1 * number2
```

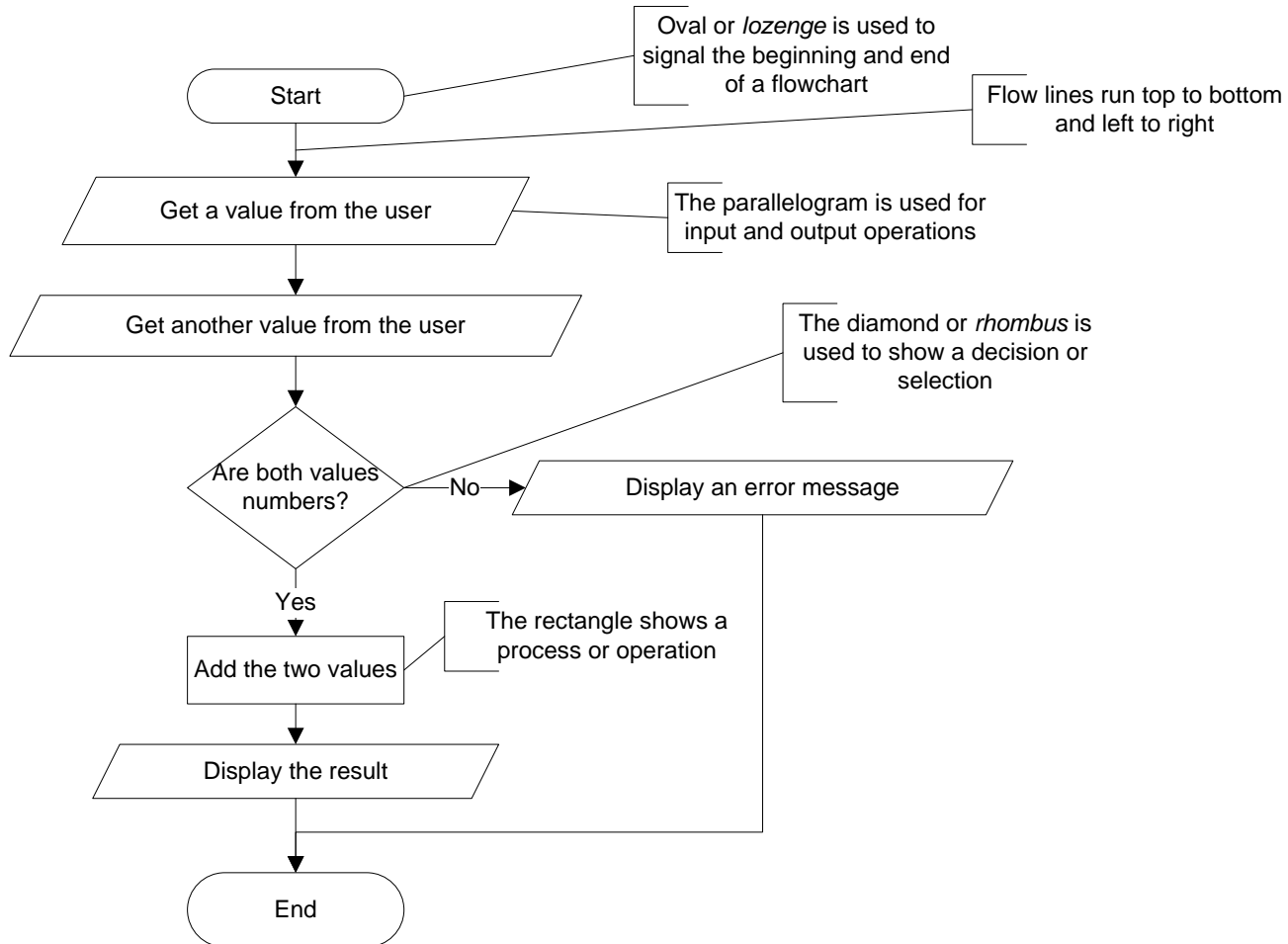
For output, we typically use the word *display* if we’re showing the result on a computer monitor, *print* if the result is being printed, and *write*, *store* or *save* if the result is to be stored in a file. In this case, we need to display to a monitor.

```
//output
Display product
```

Drawing Flowcharts

We've seen that flowcharts and pseudocode are both useful design tools. The choice of which one to use is a matter of personal preference. A flowchart is simply a more graphical representation of pseudocode. Some people work better with pictures rather than a bunch of text.

This section provides a little more formal coverage of how to draw flowcharts. First, let's take a quick look at the various symbols and notation. I think the annotations pretty clearly cover the purpose of each symbol and, be aware, you may see quiz/exam questions on these.



Note that the beginning of the flowchart can be labeled with the method name as shown in various examples throughout the textbook or you can use the more traditional "Begin" or "Start" notation shown above. At the end or termination point, you can label the lozenge with "Return" as shown in various text examples or, you can use the more traditional *End* notation shown in the sample above. Once again, it's a matter of personal preference.

Overview

Python is a programming language that includes features of C and Java. It provides the style of writing elegant code like C, and for object-oriented programming, it offers classes and objects like Java.

Python is gaining popularity in the programming community. There are some good reasons behind this:

- Interpreted Language: Python is processed at runtime by Python Interpreter.
- Object-Oriented Language: It supports object-oriented features and techniques of programming.
- It has the features of most traditional programming languages. Concepts can be applied to C++, Java, etc.
- Easy language: Python is an easy to learn language, especially for beginners.
- Straightforward Syntax: The formation of python syntax is simple and straightforward which also makes it popular.
- Simple syntax that's easy to read
- Portable: Python codes can be run on a wide variety of hardware platforms having the same interface.
- Extendable: Users can add low level-modules to Python interpreter.
- Python is used to create web and desktop applications, and some of the most popular web applications like Instagram, YouTube, Spotify have all been developed in Python, and you can also develop next big thing by using Python.

Don't worry if it doesn't all make sense right away. It takes time, practice and lots of patience! Be patient with yourself and with others in the class. You will be given all (or at least most) of the code you need to get through.

Python Development Cycle

One of the things covered in chapter one is the concept of an *Integrated Development Environment* (IDE). An IDE provides a "one stop shop" professional programming environment where we can write code, compile, debug and run our applications. For this class, we will be using IDLE (Integrated Development & Learning Environment). This is an IDE designed specifically for Python.

Python Program Structure

Every programming language, including Python, has a certain *syntax* or *grammar* that defines the language. These are somewhat similar to various grammar and punctuation rules you learned in your English classes over the years. We'll be learning the Python syntax as we progress through the chapters in the textbook.