COMP 1012 - Computer Programming for Scientists and Engineers

Course Description

Calendar entry

(Lab Required) An introduction to computer programming suitable for solving problems in science and engineering. Students will implement algorithms for numerical processing, statistical analysis and matrix operations. May not be held with COMP 1010, COMP 1011 or COMP 1013. Prerequisite: Mathematics 40S, or equivalent. Co-requisite: MATH 1230 or MATH 1500 or MATH 1501 (or equivalent).

General Course Description

This course is an introduction to implementing simple algorithms by writing computer programs. The course does not require previous programming experience. However, many students find that if they have never written any computer programs or have never been formally introduced to algorithms before, the pace can be challenging. If you would like to learn more about the subject of Computer Science, while also picking up important programming fundamentals, you may want to consider taking COMP 1500 *before* this course.

In this course students will learn to define and use variables, functions, conditional expressions, iteration via loops, read data from files stored on a computer, and create simple data structures such as arrays, lists, sets, and dictionaries. Recursion and an introduction to object-oriented programming are also discussed. Examples and problem sets will be drawn from all the Sciences (from Physics to Statistics and everything in-between) including those of particular interest to Engineers.

Detailed Prerequisites

Before entering this course, a student should be able to:

- Evaluate arithmetic expressions, applying the rules of order of operation to basic arithmetic operators (+, −, ×, ÷) and parentheses.
- Use common mathematical functions such as absolute value, square root, power functions, and trigonometric functions (e.g., to calculate angles).
- Given a formula for the terms of a sequence, calculate the sum of n terms.
- Calculate simple statistics on a dataset (e.g. mean, median, standard deviation).
- Download, install, and use new unfamiliar software on a desktop or laptop computer.

Course Goals

By the end of this course students will:

- Have a robust mental model for how the computer executes programming instructions, and makes execution flow decisions, by accessing information from memory, performing calculations, and storing results in memory.
- Write and run moderately complex programs using a procedural programming language.
- Devise solutions to simple problems and implement them as computer programs.
- Read and evaluate written programs.
- Describe basic programming concepts and structures in plain English.
- Represent ideas and information in a way that computers can understand and act on.
- Implement and use data structures to solve a problem, with emphasis on arrays, lists, sets, and dictionaries.
- Write software that reads and processes data from files stored on disk.

Learning Outcomes

The Mechanics of Programming

Students should be able to:

1. Write and edit code in a text editor or simple IDE.
2. Run their programs, providing interactive input and producing output.
3. Explain the role of an interpreter in executing a program.
4. Find and correct errors that prevent a program from running.
5. Determine the source of run-time errors using simple debugging techniques such as “trace” output statements.
6. Describe the step-by-step execution of a simple program without the use of a computer.
7. Import libraries/modules and use functions and constants defined therein.
8. Apply programming standards, such as naming conventions, commenting, and code formatting, to produce human-readable and modifiable programs.

Data and Representation

Students should be able to:

1. Identify the data types of literal values.
2. Declare, initialize, and assign variables of primitive (integer, floating point, boolean), string, sequence (lists, tuples, arrays) and collection (sets, dictionaries) types.

3. Apply operators and parentheses to build expressions using variables and literal values.
4. Determine the order of operations and the type of the result of an expression.
5. Explain the consequences of numbers having limited precision when being represented in binary by a computer.
6. Describe the scope of variables and the uses of variables of different scope.
7. Identify the difference between references and objects.

Mathematical Operations

Students should be able to:

1. Write and evaluate arithmetic expressions (+, −, ×, ÷, mod) that include both literal values and variables.
2. Apply the modulo operator in common operations such as even/odd or "clock" math.
3. Use the compound assignment operators (e.g., +=) to simplify common assignment statements.
4. Use language libraries for evaluating common math functions.

Boolean Operations

Students should be able to:

1. Use relational operators on primitive types to produce Boolean results.
2. Write and evaluate Boolean expressions with relational (e.g., >, <, >=) and/or logical operators (e.g., and, or, not).
3. Recognize the correspondence between Boolean conditions and Boolean variables.
4. Explain the order of operations used in evaluating conditional expressions.

Conditional Statements (if-else)

Students should be able to:

1. Write code that uses if and if-else constructions for decision making.
2. Write code that uses nested if statements and if-else-if chains for making more complex decisions.

String Processing

Students should be able to:

1. Declare and use string variables in a program.
2. Access characters in a string by index and build strings character-by-character or by slicing.

3. Perform simple operations on strings such as concatenation, and find and replace.
4. Convert between numeric and string types.
5. Compare strings for equality.
6. Apply formatting to strings and numbers to produce human-readable output.

Input

Students should be able to:

1. Explain why all input comes into the system as a string.
2. Obtain text input from a user.
3. Read the contents of a file.
4. Build an appropriate data structure to represent data in a file.
5. Load data from a file into a data structure using split, strip, and cast operations.

Loops

Students should be able to:

1. Write code that uses deterministic for loops, and non-deterministic loops with while.
2. Select the appropriate type of loop to use for a given context.
3. Write code that uses nested loops, and other nested control structures.
4. Use a loop to access all or a subset of elements in a data sequence.
5. Use a loop to read data from file.

Methods or Functions

Students should be able to:

1. Subdivide complex problems into subroutines.
2. Implement subroutines with parameters and return values.
3. Explain how the use of functions affects variable scope.
4. Explain the difference between passing as value and passing as reference.

Random Numbers

Students should be able to:

1. Explain why pseudorandom numbers are used in programming languages.
2. Use pseudorandom numbers to create a reproducible experiment.

Data Sequences and Collections

Students should be able to:

1. Explain the difference between a sequence (e.g. list, tuple) and a collection (e.g. set, dictionary).
2. Insert data into lists, tuples, sets, and dictionaries.
3. Access individual items or iterate over all items in a list, tuple, set, or dictionary.
4. Draw how collections of data are stored in memory.
5. Analyze a data set, and choose the most appropriate data structure to store the data.

Arrays

Students should be able to:

1. Determine when an array is an appropriate solution to a problem.
2. Explain the difference between a list and an array.
3. Declare and iterate through one-dimensional and multi-dimensional arrays of primitive types.
4. Perform a computation on, or apply a conditional to, all elements of an array.
5. Calculate basic statistics (sum, mean, median, standard deviation) for data stored in an array.
6. Pass arrays to and return arrays from subroutines.
7. Describe how arrays are passed to subroutines as variable parameters.

Object-Oriented Programming Basics

Students should be able to:

1. Write a simple class that includes constructors, instance variables, and instance methods.
2. Use instances of user-defined classes in other user-defined classes and within the main program.
3. Understand object references and use them appropriately in code, including use of a reference to the current object.
4. Analyze a problem statement or dataset and create an object or objects to represent the data.

Recursion

Students should be able to:

1. Create and implement recursive solutions to simple problems such as simple mathematical calculations and list traversals.
2. Write a recursive solution to a problem with a helper function.
3. Identify and explain the base case and recursive step components of a recursive algorithm.
4. Explain the flow of control during recursive function calls, including the role of the runtime stack.

Algorithms

Students should be able to:

1. Implement simple algorithms in a high-level programming language.
2. Devise algorithms to solve simple problems, such as array comparison or finding a minimum value.
3. Describe and implement linear search and ordered insert algorithms.