COMP 1000 – Introductory Programming: Think Like a Computer

Course Description

Calendar entry

(Lab Required) To develop the apps we use every day, you need to understand how computers 'think.' In this course students will learn to mentally simulate how a computer operates and read and write simple computer programs. Students will gain an understanding of how information is stored and computations are performed. This is an excellent pathway into computer programming for those with no prior experience.

May not be used to fulfill computer science requirements in a Computer Science Honours, Joint Honours, or Major program. May not be taken once in a declared Computer Science Honours, Joint Honours, or Major program. May be used as an elective if taken prior to entry.

Prerequisite: One of any 40S Mathematics (50%), MATH 1018, or MSKL 0100.

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, and power functions.
- Download, install, and use new unfamiliar software on a desktop or laptop computer.

Course Goals

By the end of this course students will:

- Be able to understand how a computer runs programs and will be able to examine program code.
- Simulate a computer executing simple program instructions and accurately predict the output.
- Write simple modular programs to store data, perform calculations, make decisions, repeat tedious work and respond to user input.

- Develop an appreciation for the different roles of the programmer, the program, and the end user.

Learning Outcomes

Computers, Programmers and Users

Students should be able to:

1. Describe and navigate hierarchical file systems.
2. Differentiate between software programs, programming languages, and compilers.
3. Explain the purpose of programming languages and compilers in the creation of software programs.

The Notional Machine: Storage

Students should be able to:

1. Describe how computers store things through the use variables of different data types (numbers, letters, strings, etc.).
2. Write program instructions that assign information to be stored, that retrieve stored information and that perform simple calculations or manipulations.

The Notional Machine: Output

Students should be able to:

1. Describe how computers communicate the outcome of calculations through the console and talk to end users through the screen.
2. Write simple programs that report results on the console and generate output to the screen for an end user.
3. Explain the difference between output for a programmer versus output for an end user.

The Notional Machine: Decision Making

Students should be able to:

1. Describe how computers investigate data and use that data to make decisions.
2. Apply Boolean logic and truth tables to define 'reasoning'.
3. Draw flow charts to plan how a computer should make a decision.
4. Write programs using if statements to implement flow charts.

The Notional Machine: Input

Students should be able to:

1. Describe how computers get information from the world through mouse and keyboard input.
2. Write programs that receive spatial input from users through mouse clicks and drags.
3. Write programs that receive text input from users through the keyboard.
4. Explain the difference between a programmer providing hard-coded input in a program versus an end user providing input while a program is running.

The Notional Machine: Repetition

Students should be able to:

1. Describe how computers do the same thing over and over again through the use of a loop.
2. Write programs that use loop structures to enable computers to repeat tasks without having to retype the same instructions over and over.

The Notional Machine: Functions

Students should be able to:

1. Describe how computers do lots of different things at different times through the use of named functions that can be called upon as needed.
2. Break a problem into sub-problems to design a program.
3. Create a named function, containing a set of instructions, to solve a sub-problem.
4. Write programs that contain separate named functions that can be re-used as needed.