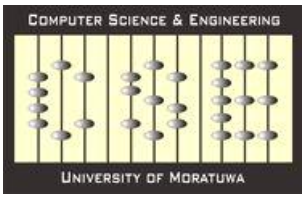



# CS1033 Programming Fundamentals

# Laboratory Exercises

BSc Engineering - Semester 1 (2021 Intake)

August – December 2022

Department of Computer Science and Engineering

Faculty of Engineering

University of Moratuwa

Sri Lanka

**Compiled by:**

Sanath Jayasena, Chandana Gamage, Gayashan Amarasinghe, Kalana Wijegunarathna, Piyumal Demotte, Haritha Jayasinghe, Ravin Gunawardena, Tarindu Jayatilaka, Thirasara Ariyarathna, Srikandabala Kogul, Kirishnni Prabagar, Budvin Edippuliarachchi and Himashi Rathnayake.



**Acknowledgements**

All help and support provided to make this document possible are acknowledged.
Sections of this document were based on:

- Python Tutorial by https://www.tutorialspoint.com/
- Introduction to Computing Using Python: An Application Development Focus by Ljubomir Perkovic.
- Python documentation at https://docs.python.org/3/contents.html
- C Programming Lab Exercises document for CS1033 used in previous years, contributed by several members of the Dept of Computer Science & Engineering. They had the following references:
    - The C Programming Language (ANSI C Edition) by Brian W Kernighan and Dennis M Ritchie, Prentice-Hall
    - 'Learn C in Three Days' by Sam A. Abolrous, BPB Publications & Programming in ANSI C, second edition by E. Balagurusamy, Tata McGraw-Hill.
- Programming problems from Instructors in the Dept of Computer Science & Engineering

**Date of this version: 31 July 2022**

# Table of Contents

# Pre-requisites

The following are assumed (these are prerequisites for CS1033 labs).

1. You have an account on the LearnOrg system (https://lms.mrt.ac.lk/) and you have logged on to it at least once.
2. You have registered for Semester 1 course modules on LearnOrg.
3. You are familiar with the LearnOrg system.
4. You have logged on to the LearnOrg-Moodle system (https://online.mrt.ac.lk/) at least once.
5. You have basic Moodle skills as a student (can access resources; can complete activities like quizzes, forums, assignments; can manage your profile).

# Pre-Lab Activity: Introduction to CS1033 Labs

(To be completed during the first lab session, before starting the CS1033 Labs starting from the next section).

---

### 1. Objectives

After completing this part successfully, you should be able to:

1. List / describe rules and student responsibilities applicable to CS1033 lab sessions.
2. Identify the staff associated with CS1033 lectures and labs.
3. Use the operating system on your personal computer with the graphical interface and with the command line for simple tasks.
4. Use python and Jupyter Notebook on your personal computer.

### 2. Procedure

**Part 1**: A staff member in the Department of Computer Science & Engineering (CSE) will help you to achieve Objectives (1) and (2) above, as follows:

- Welcome you to the Department's S1 Lab and to CS1033 lab sessions.
- Introduce the staff associated with CS1033.
- Explain relevant rules, guidelines, and your responsibilities.

**Part 2**: You will be guided by staff and other resources to achieve Objectives 3 and 4 above to get familiarized with the execution environment and tools. If you feel online interactive mode is difficult, do not worry; more practice and spending more time (as you will be doing, throughout the semester), will help. You could always seek help from staff if you need.

# Lab 1: Introduction to Python Programming

**Objectives**

- Edit a Python program.
- Execute Python programs on your preferred operating system using the Python Interpreter.
- Correct (fix) syntax errors and bugs, taking clues from error messages and warnings.
- Debug programs through program testing.
- Detect programming errors through program testing.
- Be familiar with a text editor on your preferred operating system.

**Requirements**

- The Python Interpreter (Python version 3).

**Remarks**

- This lab exercise will take more than one lab session (timetable slot).
- Lab 1 will not be evaluated.
- All the source codes and executables related to this lab session should be saved in a folder named **Lab1.**
- All the source codes related to this lab should be submitted to the respective lab activity on Moodle.

## L1.1 Using Python in Interactive Mode

To start the Python interpreter in *interactive mode*, type the command **`python`** at the command prompt or terminal.

When commands are read from the keyboard, the interpreter is said to be in interactive mode. In this mode it prompts for the next command with the primary prompt, usually three greater-than signs (>>>); for continuation lines it prompts with the secondary prompt, by default three dots (...). The interpreter prints a welcome message stating its version number and a copyright notice before printing the first prompt:

```
Python 3.10.0 (tags/v3.10.0:b494f59, Oct  4 2021, 19:00:18) [MSC v.1929
64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

After that, you can type commands and statements at the primary prompt. Some examples:

```
>>> 1 + 5
6
>>> 2 * 5
10
>>> len('aaa') + len('ttt')
6
>>> print("Hello World!\n")
Hello World!

>>> for i in range(3):
...   print(i, end=" ")
0 1 2
```

You can try many others. The interactive mode makes it easy to experiment with features of the language, or to test functions during program development. It is also a handy desk calculator.

But our objective here is to focus on the non-interactive use of Python, that is, to develop and run *Python programs* (also called *Python scripts*), which are sequences of Python commands.

### L1.2 Python Programs

Perform the following steps.

Step1: Type the Python program given below using a text editor such as **gedi**t (or **Notepad** or **vi** or **emacs** or **Kwrite**) and then save it as **test1.py** in an appropriate directory.

```
# display Hello World

print("Hello world !\n")
```

Step 2: Open a new terminal or a command window. Now you can enter any command using the shell. Go to the directory where you saved the program source file **test1.py** using **cd** command.

Step 3: Next, to run the program saved in Step 1, enter the following command at the command/shell prompt.

```
$ python test1.py
Hello world !
```

**Exercise**: Modify the above program to display the program text (source code).

*Hint:* Study the use of escape sequences (escape codes) **\n**, **\t** and \" with `print`**.**

### L1.3 Developing Programs

As described on the Section 1.1.1 of the Course Notes Part A, developing a program involves the following steps most of which are common to any problem solving task.

1. Define the problem
2. Outline the solution
3. Develop an algorithm
4. Test the algorithm for correctness
5. Code the algorithm in a programming language to obtain a program
6. Ensure program has no syntax errors
7. Compile to generate translated code (optional)
8. Run the program
9. Test and debug the program
10. Document and maintain the program

The first four steps of identifying the problem and coming up with the algorithm is crucial for the success of the solution. Once the algorithm has been outlined and tested, any suitable programming language can be chosen to implement the algorithm. A successful programmer follows these steps to develop successful solutions using programming.

As an example, let's consider how solving a quadratic equation should be approached.

Step 1: Define the problem – Develop a program that can solve a quadratic equation $ax^2 + bx + c = 0$ for `x` given the integers `a, b,` and `c`.

Step 2: Outline the solution – The solutions of a quadratic equation $ax^2+bx+c=0$ is given by the formula,

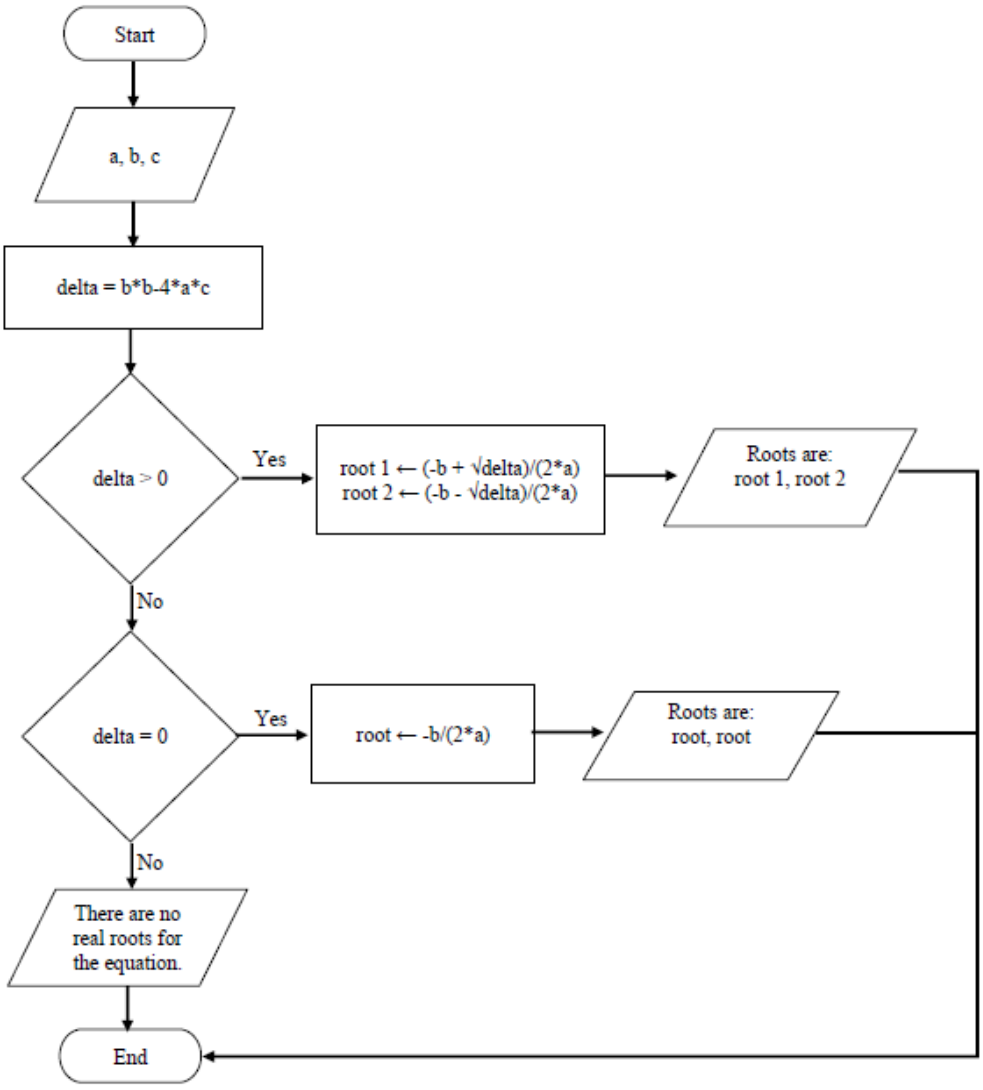
$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

provided that $b^2 - 4ac \geq 0$.

Step 3: Develop an algorithm – Let's develop the algorithm using pseudocode.

```
BEGIN
      INPUT a, b, c
      delta ← b*b -4*a*c
      IF delta > 0 THEN
            root1 ← (-b+ √delta) / (2*a)
            root2 ← (-b- √delta) / (2*a)
            OUTPUT "Roots are:", root1, root2
      ELSE IF delta = 0 THEN
            root ← -b / (2*a)
            OUTPUT "Roots are:", root, root
      ELSE
            OUTPUT "There are no real roots for the equation."
      ENDIF
END
```

Another important tool for developing algorithms is flow charts (introduced in the section 1.1.4 of Course Notes Part A). Below is the same algorithm designed using a flowchart.



Step 4: Test the algorithm for correctness – Consider a few quadratic equations that and try them with your algorithm designed in the Step 3. Check if it produces the expected answers for different scenarios.

*Hint: You can improve your algorithm during testing. See if you can improve the above algorithm.*

Step 5: Code the algorithm in a programming language to obtain a program – Now that you have a tested algorithm, you are free to choose a suitable programming language to implement your solution.

For example, If the python programming language was chosen, below is how the above algorithm would be implemented:

```
import math

a = int(input("Enter a :"))
b = int(input("Enter b :"))
c = int(input("Enter c :"))
delta = b*b - 4*a*c
if delta > 0:
    root1= (-b - math.sqrt(delta)) / (2*a)
    root2= (-b + math.sqrt(delta)) / (2*a)
    print("Roots are: ", root1, root2)
elif delta == 0:
    root = -b/(2*a)
    print("Roots are: ", root, root)
else:
    print("There are no real roots for the equation.")
```

If the C programming language was chosen, below is how the above algorithm would be implemented:

```
#include <stdio.h>
#include <math.h>
int main( )
{
        int a, b, c, delta;
        float root1, root2, root;
        printf("Enter a: ");
        scanf("%d", &a);
        printf("Enter b: ");
        scanf("%d", &b);
        printf("Enter c: ");
        scanf("%d", &c);
        delta = b*b - 4*a*c;
        if (delta > 0)
        {
            root1 = (-b - sqrt(delta)) / (2*a);
            root2 = (-b + sqrt(delta)) / (2*a);
            printf("Roots are: %.1f %.1f\n", root1, root2);
        }
        else if (delta == 0)
        {
            root = -b/(2*a);
            printf("Roots are: %.1f %.1f\n", root, root);
        }
        else
        {
            printf("There are no real roots for the equation.");
        }
        return 0;
}
```

Steps 6,7,8,9 and 10 – This lab note, and the Course Notes Part A and B will guide you through with these steps.

### L1.4 Syntax Errors and Bugs

Programmers, being humans, often end up writing source code with errors.

Some of these errors result in source code that does not conform to the *syntax rules* (*grammar rules*) of the Python language and are caught by the interpreter. Some others however, surface only later and cause programs to fail at performing their intended tasks. Such defects in programs are called *bugs* and the process of eliminating them is called *debugging*.

Perform the following steps.

Step 1: Shown below is a Python program intended to compute the roots of the quadratic equation $ax^2+bx+c=0$ provided that $b^2-4ac\geq 0$, given integers *a*, *b* and *c*.

Type the following program exactly as given using a text editor and then save it as **test2.py**.

```
a = int(input("Enter a :))
b = int(input("Enter b :"))
c = int(input("Enter c :"))
delta = b*b - 4*c
if delta < 0:
print("complex roots !")
else:
    t= math.sqrt(delta)
    r1= (-b -t) / (2*a)
    r2= (-b +t) / (2*a)
    print("Roots are: ", r1, r2)     # display the roots
```

Step 2: Now, to run the program, enter the command **python test2.py** at the command/shell prompt. The text below shows something like what you will get next!

```
$ python test2.py

File "test2.py", line 1
    a = int(input("Enter a :))
                  ^
SyntaxError: unterminated string literal (detected at line 1)
```

The above message is reported by the Python interpreter when it processes the source file, **test2.py**. This is because of the identified syntax error in the source code.

The way errors are reported depends on the interpreter and the environment. Notice that the interpreter reports the file name and the line number where it found the syntax error.

Step 3: Let's correct the error reported above. Notice that the error message is concerned with line 1.

Open the file **test2.py** in an editor and go to line 1. Try to figure out what is wrong and correct it. You may also want to take into consideration the following part of the error message:

```
unterminated string literal (detected at line 1)
```

After fixing line 1, run **test2.py** again.

If you corrected the error on line 1 properly, you should now get a message similar to the following:

```
File "test2.py", line 6
    print("complex roots !")
    ^
IndentationError: expected an indented block after 'if' statement on line 5
```

This is due an indentation error on line 6. In Python, a block of code belonging to an **if** clause should be indented to the right (e.g., with a tab). Correct this error.

After correcting line 6, go back and try to run **test2.py** again. If you correctly followed all the above instructions, you should not get any more error messages.

Step 4: You may run the program by entering the following command at the prompt. Press **Enter** key after typing the values for the coefficients **a, b,** and **c.**

```
$ python test2.py
Enter a : 1
Enter b : -7
Enter c : 12
```

Now you will see the following error message.

```
Traceback (most recent call last):
  File "test2.py", line 8, in <module>
    t= math.sqrt(delta)
NameError: name 'math' is not defined
```

Here the interpreter has reported an error. The cause is the **math.sqrt** function. Whenever any mathematical function (e.g., sqrt, sin, cos) are used, the **math** module needs to be imported. So, the above error can be corrected by inserting an "`import math`" statement at the beginning of your program as follows.

```
import math
a = int(input("Enter a :"))
b = int(input("Enter b :"))
c = int(input("Enter c :"))
....
```

Now run the program again. Press **Enter** key after typing the values for **a, b,** and **c.**

```
$ python test2.py
Enter a : 1
Enter b : -7
Enter c : 12
Roots are:  3.0 4.0
```

Here 3.0 and 4.0 are the correct roots for the given values for a, b and c.

Try the program with several more sets of values for **a, b,** and **c**. Such sets of values used for testing a program are known as *test cases*.

Step5: When developing a program, syntax errors are probably not that difficult to correct. Referring to the previous step, did the program work correctly? If you tried a test case where **a** is not equal to 1, you would notice that it gives wrong answers. Here is an example:

```
$ python test2.py
Enter a : 2
Enter b : -14
Enter c : 24
Roots are:  1.0 6.0
```

Step 6: Had you paid attention to the expression that computes the value of `delta`, you would have noticed a fault. The expression calculates the discriminator $(\Delta = b^2 - 4ac)$ of the quadratic equation. However, it fails to use the proper formula in calculation. Here is the expression:

```
delta = b * b - 4 * c;
```

This is an example of a programming error or *bug*. Detecting bugs in programs can be hard, though it is easy in this case. *Code inspection* (where one or more people read someone else's program line by line) and *program testing* (running the program on sample input data - like we did above) are two common techniques in software engineering.

Now, go ahead and correct the expression. Then run the program again with the test case for which the program failed earlier and see whether it works correctly now.

Step 7: Next, try the following test case against your program and see how it behaves when **a** is equal to zero.

```
$ python test2.py
Enter a : 0
Enter b : -7
Enter c : 12
Traceback (most recent call last):
  File "test2.py", line 10, in <module>
   r1= (-b -t) / (2*a)
  ZeroDivisionError: float division by zero
```

When a program performs an illegal operation at *run-time* (while it is executing), it is called a *run-time error*. Here, the program performed a division by zero for the input data set **{0, -7, 12}**. Looking at the source code, it is clear that if we enter a value of 0 for **a**, this leads to a division by zero later in the program. One possible solution is to check the value we input for coefficient **a** using an **if** clause. Then, since we check before we divide, even if we enter a value of zero, it does not lead to a division by zero.

Step 8: See below for the program which addresses all the above issues. Test the program with all the input combinations that we used before.

```
import math
a = int(input("Enter a :"))
b = int(input("Enter b :"))
c = int(input("Enter c :"))

if  a==0:
    print("Coefficient a should be non-zero.\n"   )
else:
    delta = b*b - 4*a*c
    if delta < 0:
        print("complex roots !")
    else:
        t= math.sqrt(delta)
        r1= (-b -t) / (2*a)
        r2= (-b +t) / (2*a)
        print("Roots are: ", r1, r2)   # display the roots
```

Step 9: Open Lab 1 quiz on the Moodle and copy the python program to the text editor and check your answer.

### L1.5 Comparing Python and C

To get a basic understanding on the differences between the Python and C languages (as well as their language implementations), we will do the following. You should first read Section 1.4 in the Course Notes. Next perform the following steps.

As you are attempting this lab on your personal computer, we do not expect you to install a C compiler. Therefore, we will use the Lab 1 quiz activity on Moodle to compare the two languages.

Step 1: Open the Lab 1 quiz on the Moodle and type in the "Hello, World!" C program in Section 1.4 in Course Notes. Check the program and see if it passes the test case.

Step 2: In the next question, type in the following C program that computes the roots of the quadratic equation $ax^2+bx+c=0$ provided that $b^2-4ac\geq 0$, given integers *a*, *b* and *c*. This C program was developed by referring to the Python program **test2.py** above of this lab. Check the program and see if it passes the test case.

```
#include <stdio.h>
```

```
#include <math.h>
int main( )
{
        int     a, b, c, delta;
        float   t, r1, r2;
        printf("Enter a: ");
        scanf("%d", &a);
        printf("Enter b: ");
        scanf("%d", &b);
        printf("Enter c: ");
        scanf("%d", &c);
        if  (a==0)
                printf ("Coefficient a should be non-zero.\n");
        else {
                delta = b*b - 4*a*c;
                if (delta < 0)
                        printf("complex roots !\n");
                else {
                        t = sqrt(delta);
                        r1 = (-b - t) / (2*a);
                        r2 = (-b + t) / (2*a);
                        printf("Roots are: %.1f %.1f\n", r1, r2);
                }
        }
        return 0;
}
```

Step 3: What are the main differences between the two languages and how they are implemented? Based on the Section 1.4 of the lecture notes, the C program is compiled into an executable code and stored on disk. What do you say about the lengths of the C program and the Python program, intended for the same task?

If you are interested in compiling a C program, you can try the following steps to compile and run a C program on your computer. You will have to have installed a compatible C compiler on your operating system.

Optional Step 1: Save the above C program using a text editor and save it as **test2.c**. Compile the C program in **test2.c** file as follows. This will produce the executable file **a.out**, if there are no compilation or linking errors. (Try compiling without "`-lm`" and see).

```
$ gcc test2.c -lm
```

Optional Step 2: Run the executable as follows:

```
$ ./a.out
```

### L1.6 Practice Programs

1. To get more understanding on the basics of Python, practice commands and programs in Chapter 2 of the CS1033 Course Notes.
2. To get more understanding on Operators and Expressions in Python, practice commands and programs in Chapter 3 of the CS1033 Course Notes.

# Lab 2: Conditional Control Structures

**Remarks**

- This lab will not be evaluated.
- Express each solution as a pseudocode and a flowchart.
- For drawing flowcharts, you can use an online tool such as lucidchart or you can draw on a paper and upload the photograph of the diagram.
- All the source codes and diagrams related to the Exercises L2.E1, L2.E2 and L2.E3 should be submitted to the respective lab activity on Moodle.

**L2.1 Practice Programs**

1. To get a better understanding on Conditional Control Structures, practice programs in Chapter 4 of the given CS1033 Course Notes.

**L2.2 Exercises**

***Exercise L2.E1*** **-** Develop a program to input three angles of a triangle and output the type of the triangle. First you have to check whether the given angles form a triangle or not. If they do form a triangle then output the type of the triangle as "Right angled", "Obtuse angled" or "Acute angled" triangle. If the angles do not form a triangle you can output "Angles do not form a triangle". You must draw a flowchart to express the algorithm of your program.

Input:
```
Enter angle 1: 45
Enter angle 2: 90
Enter angle 3: 45
```
Output:
```
Right angled
```

***Exercise L2.E2*** **–** Given a date as a triplet of numbers **(y, m, d),** with **y** indicating the year, **m** the month (m = 1 for January, m = 2 for February, etc.), and **d** the day of the month, the corresponding day of the week **f** (f = 0 for Sunday, f = 1 for Monday, etc), can be found as follows:

a) if m < 3 let m = m + 12 and let y = y - 1
b) let a = 2m + 6 (m + 1) / 10
c) let b = y + y/4 – y/100 + y/400
d) let $f_1$ = d + a + b +1
e) let f = $f_1$ mod 7
f) stop

Develop a program that reads a date as three numbers separated by white spaces and outputs the corresponding day of the week. All divisions indicated above are integer divisions. You must draw a flowchart to express the algorithm of your program.

***Exercise L2.E3*** **–** Develop a program to input the units of electricity (number of kWh) consumed monthly and output the calculated electricity bill for that month. The charges differ according to the following domestic purpose plan from Ceylon Electricity Board.

- If the consumption is between 0-60 kWh per month the following tariffs will be applicable.

| Monthly Consumption kWh | Unit Charge (Rs/kWh) | Fixed Charge (Rs/month) |
|---|---|---|
| 0-30 | 2.50 | 30.00 |
| 31-60 | 4.85 | 60.00 |

- If the consumption is above 60 kWh per month the following tariffs will be applicable.

| Monthly Consumption kWh | Unit charge (Rs/kWh) | Fixed charge (Rs/month) |
|---|---|---|
| 0-60 | 7.85 | - |
| 61-90 | 10.00 | 90.00 |
| 91-120 | 27.75 | 480.00 |
| 121-180 | 32.00 | 480.00 |

| | | |
|---|---|---|
| >180 | 45.00 | 540.00 |

***e.g -*** If the consumption is 45kWh, the calculation would be as follows:

Monthly Bill = 30 kWh * 2.5 Rs/kWh + (45-30) kWh * 4.85 Rs/kWh + Rs 60 = Rs 207.75

Note that the fixed cost in the calculation (Rs 60 in the above example) is the fixed cost pertaining to the total consumption. The fixed cost is not cumulative.

# Lab 3: IDEs and Debuggers

**Objectives**

- This lab will not be evaluated.
- Develop Python programs using an *Integrated Development Environment* (IDE).
- Correct syntax errors and bugs by taking clues from error messages.
- Use a debugger to debug a program.

**Prerequisites**

- Students are expected to be familiar in developing Python programs on their preferred platform.

**Requirements**

- The *Python IDLE* IDE.

**Remarks**

- This lab will not be evaluated.
- All the source codes and executables related to this lab session should be saved in a folder named **Lab3**.
- All the source codes related to the lab should be submitted to the respective lab activity on Moodle.

*Integrated Development Environment* (IDE) is a Graphical User Interface (GUI) based application or a set of tools that allows a programmer to write, compile/interpret, run, edit and in some cases test and debug within an integrated, interactive environment. These tools provide a lot of features that make programming a fun activity. It may take some time for you to understand each and every feature provided by the IDE but it will certainly enhance your productivity. Some of these tools go beyond conventional program development and testing and even provide facilities to manage various documents related to the project or source code to keep track of different versions of the source code. *Python IDLE* is a popular IDE for developing Python programs. From this lab session onward, you can use Python IDLE for program development.

### L3.1 Using Python IDLE

To launch Python IDLE, type the command `idle` at the command/shell prompt on a terminal on linux or by accessing Python IDLE through the windows start menu.

Then the Python IDLE window appears.

### L3.2 Your First Program with IDLE

You can work interactively with IDLE, the same way you would interact with the Python interpreter running on a terminal. You can also run Python programs (scripts) using this IDE. To create a new file you should follow the following sequence.

**File → New File**

Then a new editor window will appear and you can write your Python program in that editor.

Now enter the following line in the editor:

```
print("Hello world !\n")
```

Next it is time to run your Python program using IDLE. First you need to save your program. Use **File → Save** and select a folder on your computer.

Next, to run your program, either select **Run → Run Module** or press the function key **F5**.

If your program gets successfully executed you should see the following message appearing in the main IDLE window (Python Shell).

```
Hello, World !
```

### L3.3 Locating Errors in a Program

Let us write a program to compute the factorial of a given number.

Step 1: Create a new Python file named **FactIDE.py**. Then type the following program exactly as shown.

```
import math
num = int(input("Enter number to find factorial :"))
fact=0
for i in range (1, num+1):
        fact = fact*i
print("Factorical of ", num, " is ", fact)
```

Step 2: Run the program by pressing **F5** or using **Run → Run Module**.

Executing the program with 5 as the input will display:

```
Enter number to find factorial: 5
Factorial of 5 is: 0
```

You will get something similar to above. But, what is the factorial of 5? Is it 0? Although your program runs, you get the wrong output which means you have a logical error (bug). These type of errors can be tracked by debugging a program.

### L3.4 Debugging a Program

Now let us try to find a bug in the above program using a Debugger. You can refer to the video available on the Moodle course page to learn how to use the python debugger.

Step 1: Go back to the source code and place a **Breakpoint** on the line with the expression `fact =fact*i.` To place a Breakpoint move your mouse pointer to the relevant line and right click on the line and select **Set Breakpoint** from the sub menu. Then you should see the line highlighted in yellow colour. To clear a Breakpoint right click on the line again and select **Clear Breakpoint** from the sub menu.

Step 2: To debug your program select **Debug → Debugger** from the IDLE main window. Then you will see the Debug control window on the screen.

Step 3: Go back to the source code and run the program by pressing **F5** or selecting **Run → Run Module**. Then you will see the Debug control window appearing again on the screen.

Click on the **Go** button in the Debug control window to start the execution.
When the program requests you to enter a number to find the factorial, type 5 and press the Enter key. Then the program *will execute up to the breakpoint and halt*.

Step 4: In the Debug Control window you will see four checkboxes showing the Stack, Locals, Globals and the Source.
Locals shows the current values of the local variables. Globals shows the current values of the global variables.

Step 5: Click on the **Go** button.

Notice that the value of variable i gets changed but not the values of the other variables.

Step 6: Again select **Run → Run Module**. You will still see only variable **i** is getting modified in the iterations of the for loop whereas variable **fact** should also get modified. Click on the Go button until you complete all the 5 iterations of the for loop.

Now you should be able to understand that there is some error in the statement `fact=fact*i` which is supposed to update the value of variable **fact** inside the for loop. However you did observe that only variable **i** is getting updated. Then the error should be with the value of **fact**. Do you remember that the starting value of **fact** is **0** ? If not, use the Debugger again and verify that **fact** is always **0**. So regardless of the value of **i**, the multiplication always results in **0**. This is the bug in the program.

To debug (to fix the error), change the initial value of variable **fact** to **1** and **Run** your program again. Now enter 5 and verify that you get the correct answer. Test your program with several other inputs.

# Lab 4: Loop Control Structures

**Remarks**

- **Exercises L4.E1, L4.E2 and L4.E3** in this lab session will be **evaluated**.
- First practice the programs mentioned in L4.1. These will not be evaluated, and you do not have to submit them.
- You can either use a simple text editor or an IDE such as IDLE.
- Express each solution as a pseudocode and a flowchart.
- All the source codes and diagrams related to the Exercises L4.E1, L4.E2 and L4.E3 should be submitted to the respective lab activity on Moodle.

**L4.1 Practice Programs**

1. To get a better understanding on Loop Control Structures, practice programs in Chapter 5 of the given CS1033 Course Notes.

**L4.2 Exercises**

***Exercise L4.E1*** - Develop a program that takes as input a series of positive integers and outputs whether each is a prime. The program should terminate if a negative integer is given as the input. A prime number is a number that is divisible by only 1 and itself. However, 1 is not considered a prime number. Execution of your program should produce something similar to the following:

```
$ python prime.py
1
non-prime
2
prime
3
prime
4
non-prime
5
prime
-1
```

***Exercise L4.E2*** - In number theory, an *abundant number* is a number for which the sum of its proper divisors is greater than the number itself. The integer 12 is the first abundant number. Its proper divisors are 1, 2, 3, 4 and 6 which sums up to 16. So, the total is greater than the number 12. The integer 14 is not an abundant number. Its proper divisors are 1, 2 and 7 which sums up to 10. So, the total is not greater than the number 14.

Develop a Python program to take as input a positive integer *n* greater than 1 and output the number of abundant numbers from 2 to *n* inclusive. Output 'Invalid Input' for inputs less than 2.

Input:

```
Input number: 15
```

Output:

```
Number of abundant numbers from 1 to 15 is 1
```

***Exercise L4.E3*** - Encryption is the process of encoding information to prevent anyone other than its intended recipient from viewing it. Develop a Python program to encrypt a message. The message will be encrypted using a given *base*. You must obtain the ASCII value of each character of the message and convert that number into the given base. After converting values of all the characters, aggregate (concatenate) those values in the character order and show it as the encrypted message. An example is given below.

Note: ($1 < \text{base} \leq 10$)

Input:

```
Enter message: Welcome to CSE
Enter base: 4
```

Output: `111312111230120312331231121120013101233200100311031011`

# Lab 5: Lists

**Remarks**

- **Exercises L5.E1, L5.E2, L5.E3** and **L5.E4** in this lab session will be **evaluated**.
- First practice the programs mentioned in L5.1. These will not be evaluated, and you do not have to submit them.
- Express each solution as a pseudocode and a flowchart.
- All the source codes and diagrams related to the Exercises L5.E1, L5.E2, L5.E3 and L5.E4 should be submitted to the respective lab activity on Moodle.

**L5.1 Practice Programs**

1. For basic understanding on Lists, practice programs in Chapter 6 of the CS1033 Course Notes.

**L5.2 Exercises**

***Exercise L5.E1 -*** Develop a program to find and display the minimum and the maximum among 10 numbers separated by white spaces (see the example) entered from the keyboard. You must use a list to store the numbers entered. The numbers can be negative or non-integers. An example would be as follows:

```
Input:   5   7.8   9.6   54   3.4   1.2   3   7   8.8   5
Output:  Minimum = 1.2
         Maximum = 54

Input:   4   -6   5.2   -3   4   2.4   35   734   9   -35.3
Output:  Minimum = -35.3
         Maximum = 734
```

***Exercise L5.E2 -*** Develop a program to read the names of two sports that you and your friends love to play and watch. Then generate all sentences where the subject is in ["I", "We"], the verb is in ["play", "watch"] and the object is in the two sports. Use lists to store the words and generate the sentences by iterating through the lists using deeply nested loops. An example:

```
Input:   cricket football

Output:  I play cricket.
         I watch cricket.
         I play football.
         I watch football.
         We play cricket.
         We watch cricket.
         We play football.
         We watch football.
```

***Exercise L5.E3*** Suppose there are 4 students each having marks of 3 subjects. Develop a program to read the marks from the keyboard and calculate and display the total marks and average mark (rounded off to one decimal point) of each student. Use a 2D (two-dimensional) list to store the marks. An example would be as follows:

```
Input (Enter the marks of four students, on four rows):
  50   60   80
  60   75   90
  30   49   99
  66   58   67
Output (Total marks and average mark of four students):
  Total: 190 Average: 63.3
  Total: 225 Average: 75
  Total: 178 Average: 59.3
```

```
Total: 191 Average: 63.7
```

***Exercise L5.E4 -*** Develop a program to input a matrix with any dimension and output the transpose of that matrix. You should stop accepting the rows when -1 is entered as the input. Use a 2D (two-dimensional) list to store the matrix. You should handle the exceptions such as checking the invalid rows with an inconsistent number of elements. An example:

Input:

```
1 2 3 4 5 6
2 3 4 5 6 7
3 4 5 6 7 8
4 5 6 7 8 9
-1
```

Output:

```
1 2 3 4
2 3 4 5
3 4 5 6
4 5 6 7
5 6 7 8
6 7 8 9
```

Note: Print "**Invalid Matrix**" as the error message for invalid rows with an inconsistent number of elements. Print "**Error**" for any other exceptions you are handling.

# Lab 6: Functions

**Remarks**

- **Exercise L6.E1** in this lab session will be **evaluated**.
- First practice the programs mentioned in L6.1. These will not be evaluated, and you do not have to submit them.
- Express your solution as a pseudocode and a flowchart.
- All the source codes and diagrams related to the Exercise L6.E1 should be submitted to the respective lab activity on Moodle.

**L6.1 Practice Programs**

1. For basic understanding, practice the programs in Chapter 9 of the CS1033 Course Notes.

**L6.2 Exercises**

***Exercise L6.E1 -*** Develop a program to read in two matrices A and B with dimensions n x m and compute and display their product $AB^T$ (of dimensions n x n). Assume that the elements of the matrices are integers. You must use functions while implementing this program. An example is shown below.

Note: You should check for any errors when entering the matrices. Print "**Invalid Matrix**" as the error message if the entered matrix does not comply with the given dimensions. Print "**Error**" for any other exceptions you are handling.

```
$ python matrix.py
Enter the dimension: 3,4
Enter Matrix A:
1  2  3  4
3  3  4  4
4  4  5  5

Enter Matrix B:
1  7  3  3
3  7  4  4
5  7  5  5

Matrix A X Transpose(B):
36  45  54
48  62  76
62  80  98
```

Input: A line corresponds to a row of a matrix. The elements in a row will be separated by a single whitespace. The output should be formatted similarly.

Define global variables of two-dimensional arrays to store the values of the matrices A and B. Break down the program into four functions as follows. Name the functions appropriately.

1. Read in the elements of the matrices A and B
2. Compute the Transpose of a matrix
3. Compute the product of the two matrices
4. Display the resultant matrix $AB^T$

# Lab 7: File Handling

**Remarks**

- **Exercise L7.E1** in this lab session will be **evaluated**.
- First practice the programs mentioned in L7.1. These will not be evaluated, and you do not have to submit them.
- Express your solution as a pseudocode and a flowchart.
- All the source codes and diagrams related to the Exercise L7.E1 should be submitted to the respective lab activity on Moodle.

**L7.1 Practice Programs**

1. For basic understanding, practice the programs in Chapter 10 of the CS1033 Course Notes.

**L7.2 Exercises**

***Exercise L7.E1* –** Develop a program to read the name, birthday and gender of a person from a file and output the ten-digit national identity card (NIC) number. The first four digits of the ID card is the birth year and the next three digits are the number of days to the birth date from January 1st of that year. If the person is a female, 500 is added to that value. The next 3 digits are assigned in the order of submission for a particular birth year (The input file contains the records in the order of submission where each attribute is space separated). An Example is shown below.

Input File:

```
Saman 1990-05-03 M
Aruni 1990-04-06 F
Kumaran 1988-03-05 M
Nazar 1997-09-24 M
```

Output File :

```
Saman 1990123001
Aruni 1990596002
Kumaran 1988065001
Nazar 1997267001
```

1. Skeleton of the **L7.E1** code is given in the Moodle Lab activity.
2. Develop the program using the given skeleton.

# Lab 8: File Handling and Functions

**Remarks**

- **Exercise L8.E1** in this lab session will be **evaluated**.
- The source codes should be saved in a folder named **Lab8** in your **home** directory.
- Express your solution as a pseudocode and a flowchart.

- All the source codes and diagrams related to the Exercise L8.E1 should be submitted to the respective lab activity on Moodle.

**L8.1 Practice Programs**

No special practice programs are provided. Lab 7 must be completed before attempting this lab.

**L8.2 Exercises**

***Exercise L8.E1 –*** The *Fibonacci sequence* is defined with an integer *n* as,

$$F(n) = F(n-1) + F(n-2) \text{ for } n \geq 2, \text{ where } F(0) = 0, F(1) = 1.$$

You have to develop a Python program to compute $F(n)$ for a given integer $n$ between 0 and 20.

| **Input:** | **Output:** |
|---|---|
| $0 \leq n \leq 20$ | $F(n) \leq 2^{32}$ |

Compute *F*(*n*) starting with *n*=2 and increment iteratively until the required value is obtained. Follow the instructions below.

1. Create a text file containing the value of $n$. The text file needs to be saved in the same folder where the source file has been saved. The file name will be given as input through the terminal.

2. Develop a Python function named "getNum" to read the number n from the file.

3. Develop a Python function named "show" to display the given value n and the computed value of *F*(*n*) on the screen. (Expected output format `Fibonacci(3) = 2` )

4. Develop a Python function named "saveFile" to write what was displayed in (3) above to a text file named "result.txt". This file should be in the same folder where the source file is.

5. The program should call the appropriate functions to achieve the task.

6. Print Invalid input. for values of n beyond the input range or other incompatible values.

# Lab 9a: Application of Data Structures

**Remarks**

- **Exercise L9.E1** in this lab session will be **evaluated**.
- All the source codes related to Exercise L9.E1 should be submitted to the respective lab activity on Moodle.

**Prerequisites**

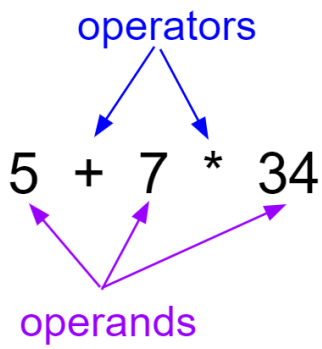
- This lab is based on lectures 8, 9, and 10. Students are expected to be familiar with the lecture content.

**L9.1 Practice Programs**

No special practice programs are provided.

**L9.2 Exercises**

***Exercise L9.E1 -*** You are required to implement a simple symbolic equation solver. The equation should be stored in a **binary tree**. An equation consists of operands and operators.




Each operand or operator should be stored as a **tuple** of the form (TYPE, VALUE). Examples are (OPERAND, 5), (OPERAND, 7), (OPERAND, 34), (OPERATOR, '+') or (OPERATOR, '*').

Following operators should be supported: addition (+), subtraction (-), multiplication (*), and exponentiation (^). Grouping of terms in the equation using brackets should be supported. However, nested brackets need not be supported. For example, the expression "1 + (2 * 3) + 3 ^ 2" should be supported but the expression "1 + ((2 * 3) + 3) ^ 2" need not be supported. In addition, the expressions will have a maximum of only one operator inside the brackets. For example, you do not have to consider expressions such as "1 + (2 * 3 + 7) + 3 ^ 2" which contains two operators '*' and '+' inside the brackets.

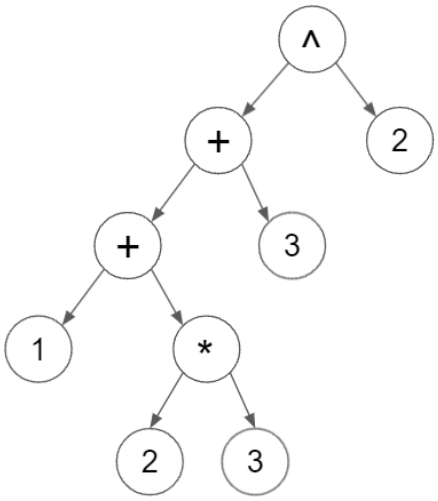
Evaluation of terms should be done **left-to-right** subjected to precedence given by brackets. (i.e. all the operators have equal precedence except the brackets). For example, the expression "1 + (2 * 3) + 3 ^ 2" will result in 100.

Explanation:

Total = 1
Total = 1 + (2*3) = 7 *(2*3 is evaluated first since they are within brackets)*
Total = 7 + 3 = 10
Total = 10 ^ 2 = 100

*Note that these are not our usual arithmetic operations where we follow the BODMAS order*.

**Hint:** Sample binary tree for the expression "1 + (2 * 3) + 3 ^ 2"




A basic template for your program is provided below.

```
class Node:

    def __init__(self, data):
        self.left = None
        self.right = None
        self.data = data

    def get_output(self):
        '''
        Print the output depending on the evaluated value.
        If the 0 <= value <= 999 the value is printed.
        If the value < 0, UNDERFLOW is printed.
        If the value > 999, OVERFLOW is printed.

        :return: None
        '''
        value = self.evaluate()
        if value > 999:
            print('OVERFLOW')
        elif value < 0:
            print('UNDERFLOW')
        else:
            print(value)

    ###########################################################################
    ######### Your task is to implement the following methods. ##########
    ###########################################################################

    def insert(self, data, bracketed):
        '''
        Insert operators and operands into the binary tree.

        :param data: Operator or operand as a tuple. E.g.: ('OPERAND', 34),
            ('OPERATOR', '+')
        :param bracketed: denote whether an operator is inside brackets or
            not. If the operator is inside brackets, we set bracketed as
            True.
        :return: self
```

```
        '''
        return self

    def evaluate(self):
        '''
        Process the expression stored in the binary tree and compute the final
            result.
        To do that, the function should be able to traverse the binary tree.

        Note that the evaluate function does not check for overflow or
            underflow.

        :return: the evaluated value
        '''
        pass
```

This template is available as lab9_template.py in the Lab 9 folder on Moodle.

Your task is to complete the body of the functions, insert and evaluate.

- The parameter data will be used to store the tuple (e.g. (OPERAND, 34), (OPERATOR, '+') ).
- The insert function is used to insert a node into the binary tree. The parameter bracketed in the insert function is used to denote whether an operator is inside brackets or not. If the operator is inside brackets, we set bracketed as True (see the examples provided at the end).
- The evaluate function should be able to process the expression stored in the binary tree and compute the final result. To do that, the function should be able to traverse the binary tree. Observe how the evaluate function is used inside the get_ouput function. For example, if the expression stored in the binary tree is "1 + (2 * 3) + 3 ^ 2", the evaluate function should return 100.
- The get_output function returns the final result (i.e., the output of the evaluate function) if it is in the range [0, 999]. If the final result is less than 0, it returns UNDERFLOW and if the final result is greater than 999, it returns OVERFLOW.
- You can create and use additional functions as required. However, you are supposed to form the tree using the insert function and evaluate the result using the evaluate function.

The following examples will give you additional information about how the functions are expected to behave. The test cases for this lab exercise are also formed similarly.

**Expression**: 1 + (2 * 3) + 3 ^ 2

```
# Use the insert method to add nodes
# Begin with forming the root node for the tree.
root = Node(('OPERAND', 1))
# Form the rest of the tree by inserting data to the root node.
root = root.insert(('OPERATOR', '+'), False)
root = root.insert(('OPERAND', 2), False)
root = root.insert(('OPERATOR', '*'), True)
root = root.insert(('OPERAND', 3), False)
root = root.insert(('OPERATOR', '+'), False)
root = root.insert(('OPERAND', 3), False)
root = root.insert(('OPERATOR', '^'), False)
root = root.insert(('OPERAND', 2), False)
# Get the output.
```

```
root.get_output()
# Should print 100
```

---

**Expression:** 1 + 2 * 3 + 3 ^ 2

```
root = Node(('OPERAND', 1))
root = root.insert(('OPERATOR', '+'), False)
root = root.insert(('OPERAND', 2), False)
root = root.insert(('OPERATOR', '*'), False)
root = root.insert(('OPERAND', 3), False)
root = root.insert(('OPERATOR', '+'), False)
root = root.insert(('OPERAND', 3), False)
root = root.insert(('OPERATOR', '^'), False)
root = root.insert(('OPERAND', 2), False)

root.get_output()
# Should print 144
```

---

**Expression:** -15 - (99 * 10)

```
root = Node(('OPERAND',-15))
root = root.insert(('OPERATOR','-'),False)
root = root.insert(('OPERAND',99),False)
root = root.insert(('OPERATOR','*'),True)
root = root.insert(('OPERAND',10),False)

root.get_output()
# Should print UNDERFLOW
```

# Lab 10a: Problem Solving

**Remarks**

- **Exercise L10.E1** in this lab session will **not be evaluated**.
- All the source codes related to Exercise L10.E1 should be submitted to the respective lab activity on Moodle.

**Prerequisites**

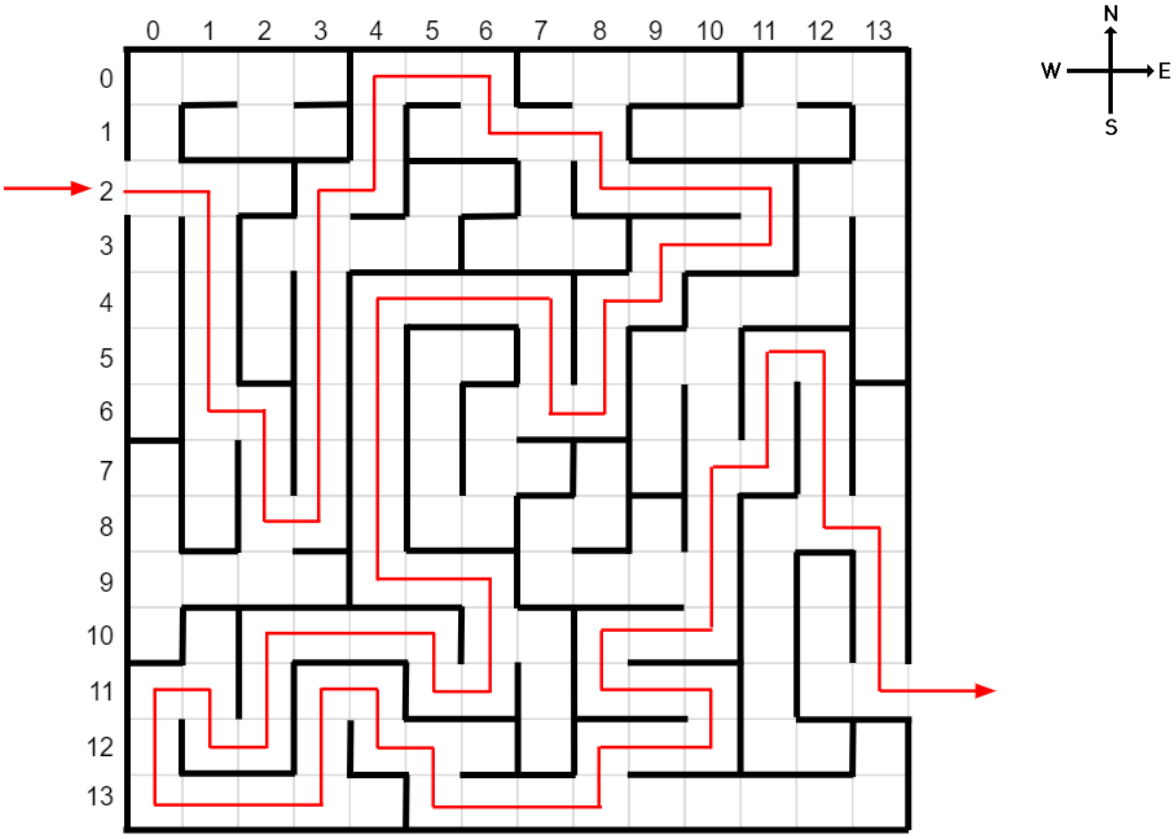
- This lab is based on lectures 8, 9, and 10. Students are expected to be familiar with the lecture content.

**L10.1 Practice Programs**

No special practice programs are provided.

**L10.2 Exercises**

***Exercise L10.E1 -*** Write a Python program to successfully navigate through the maze shown in Figure 1 below entering at cell (2,0) and exiting at cell (11,13). Your solution **must** use a matrix and a stack.




When you navigate through the maze, print the steps that you take in the following manner. An example output is shown below with explanation. Each line below ends with a new line.

**Start at ( 2 , 0 )** – *enter into the maze from the cell at (2,0) coordinates.*
**North North East East East Stuck at ( 0 , 3 )** – *starting from cell at (2,0), navigate to the North cell at (1,0). Then navigate to North cell (0,0), East cell (0,1), East cell (0,2), East cell (0,3) until you are stuck.* ***Note that the order of the directions of traversing is defined as North, East, South, West.***
**Back to ( 0 , 2 )** – *Since you could not continue from the cell at (0,3), get back to the cell at (0,2) where you can take a different path.*
**South East Stuck at ( 1 , 3 )** – *starting from the cell at (0,2), move to South cell (1,2). Then move to East cell at (1,3) where you are stuck.*
**Back to ( 1 , 2 )** – *backtrack to cell at (1,2) where you can take a different path.*
….

**Stuck at ( 7 , 13 )** – *you are stuck at the cell (7,13)*
**Back to ( 8 , 13 )** – *backtrack to cell at (8,13) where you can take a different path.*
**South South South Leaving at ( 11 , 13 ) -** *starting from the cell at (8,13), move to the South cell (9,13). Then move to the South cell at (10,13), and to the South cell (11,13) where you can leave the maze.*

The expected output after traversing the above maze is below:

```
Start at ( 2 , 0 )
North North East East East Stuck at ( 0 , 3 )
Back to ( 0 , 2 )
South East Stuck at ( 1 , 3 )
Back to ( 1 , 2 )
West Stuck at ( 1 , 1 )
Back to ( 1 , 2 )
Stuck at ( 1 , 2 )
Back to ( 0 , 2 )
Stuck at ( 0 , 2 )
Back to ( 0 , 1 )
Stuck at ( 0 , 1 )
Back to ( 0 , 0 )
Stuck at ( 0 , 0 )
Back to ( 1 , 0 )
Stuck at ( 1 , 0 )
Back to ( 2 , 0 )
East East Stuck at ( 2 , 2 )
Back to ( 2 , 1 )
South South South South East South South East North North North North North North
East North North East East South East East North East East Stuck at ( 0 , 10 )
Back to ( 0 , 9 )
Stuck at ( 0 , 9 )
Back to ( 0 , 8 )
West Stuck at ( 0 , 7 )
Back to ( 0 , 8 )
Stuck at ( 0 , 8 )
Back to ( 1 , 8 )
South East East East South West West South West South South West North North West
West West South South South South South East East South East South South Stuck at
( 12 , 7 )
Back to ( 11 , 7 )
Stuck at ( 11 , 7 )
Back to ( 10 , 7 )
Stuck at ( 10 , 7 )
Back to ( 10 , 6 )
South West North West West West South South West North North Stuck at ( 10 , 1 )
Back to ( 11 , 1 )
West South South East East East North North East South East East Stuck at ( 12 , 6
)
Back to ( 12 , 5 )
South East East East North East East North West West North East East North North
North North North North East East North North East North North West West South
East Stuck at ( 1 , 12 )
Back to ( 1 , 11 )
West West Stuck at ( 1 , 9 )
Back to ( 1 , 10 )
Stuck at ( 1 , 10 )
Back to ( 1 , 11 )
Stuck at ( 1 , 11 )
Back to ( 0 , 11 )
Stuck at ( 0 , 11 )
Back to ( 0 , 12 )
```

```
Stuck at ( 0 , 12 )
Back to ( 0 , 13 )
Stuck at ( 0 , 13 )
Back to ( 1 , 13 )
Stuck at ( 1 , 13 )
Back to ( 2 , 13 )
South South South Stuck at ( 5 , 13 )
Back to ( 4 , 13 )
Stuck at ( 4 , 13 )
Back to ( 3 , 13 )
Stuck at ( 3 , 13 )
Back to ( 2 , 13 )
Stuck at ( 2 , 13 )
Back to ( 2 , 12 )
Stuck at ( 2 , 12 )
Back to ( 3 , 12 )
Stuck at ( 3 , 12 )
Back to ( 4 , 12 )
Stuck at ( 4 , 12 )
Back to ( 4 , 11 )
Stuck at ( 4 , 11 )
Back to ( 4 , 10 )
Stuck at ( 4 , 10 )
Back to ( 5 , 10 )
West South South Stuck at ( 7 , 9 )
Back to ( 6 , 9 )
Stuck at ( 6 , 9 )
Back to ( 5 , 9 )
Stuck at ( 5 , 9 )
Back to ( 5 , 10 )
Stuck at ( 5 , 10 )
Back to ( 6 , 10 )
Stuck at ( 6 , 10 )
Back to ( 7 , 10 )
East North North East South South South East North North Stuck at ( 6 , 13 )
Back to ( 7 , 13 )
Stuck at ( 7 , 13 )
Back to ( 8 , 13 )
South South South Leaving at ( 11 , 13 )
```

# Lab 9b: Introduction to Arduino Development Environment

**Objective**
After successfully completing this lab session, you should be able to:

- Use the Arduino Integrated Development Environment (IDE)
- Develop simple embedded applications using Arduino board and IDE

**Prerequisites:** Students are expected to know basic circuit concepts and programming concepts.

**Remarks**

- This lab will not be evaluated, but what you learn in this lab will be essential for the next lab.
- This lab will be done individually.
- All the source codes in this lab should be saved in a folder named **Lab9** in the **home** directory.
- The files should be uploaded as a zip **Lab9_<IndexNo>.zip** to learnOrg-Moodle.
- Be careful when wiring; do not connect power and ground lines together at any time.
- If you need any guidance during this lab, ask for the support from an instructor.
- Before leaving the lab, show your outcomes to the instructor.

## Part 1 – Introduction to Arduino Environment

**Required Hardware Components**

- Arduino board
- Breadboard
- Computer which is installed with Arduino IDE
- USB cable (to transfer instructions and data between PC and Arduino board)
- LED – Light Emitting Diode
- 330Ω Resistor
- Prototype Wires

***Task 1*: Preparing the circuit to blink an LED**
Use the given components and build the circuit that we will use to blink an LED as follows. Connect the positive pin of the LED to pin 10 as indicated though the breadboard. The resistor (330Ω) in series is to limit the current through the LED (Fig. 1). Connect the negative pin of the LED to ground (GND) pin of the board. Now the circuit for the Part 1 of the lab is ready.

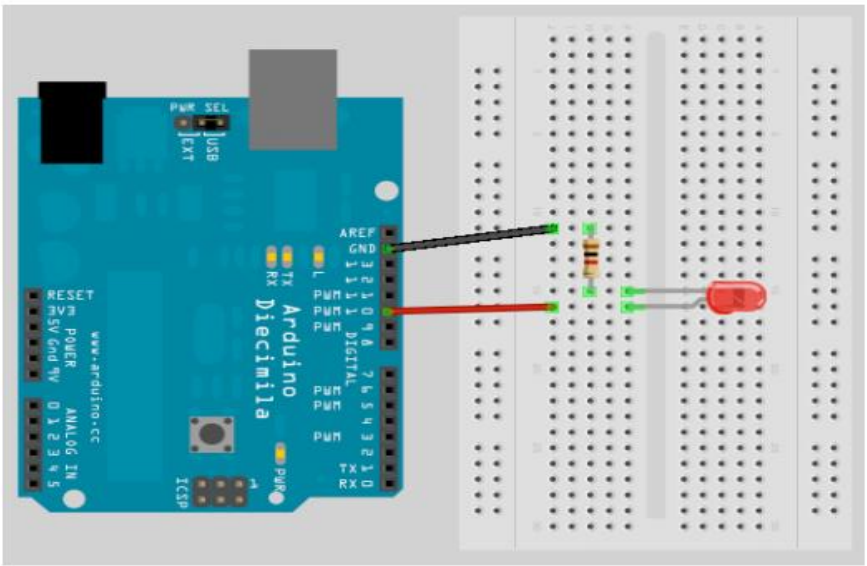



*Figure 1: Connection of the LED and resistor to the Arduino board.*

***Task 2*: Writing a basic program with Arduino IDE**
Open the Arduino IDE (Arduino.exe). This will open a blank file on Arduino IDE.

You can identify the main areas of the Arduino IDE as shown in the Fig. 2. The Program Editing Area is to write the code. The tool bar consists of several buttons like a button to Compile/ Verify the code to see if there are any syntax errors of the code, an Upload button to upload the code to the Arduino board and a button to open Serial Monitor which can be used to communicate with the Arduino board.

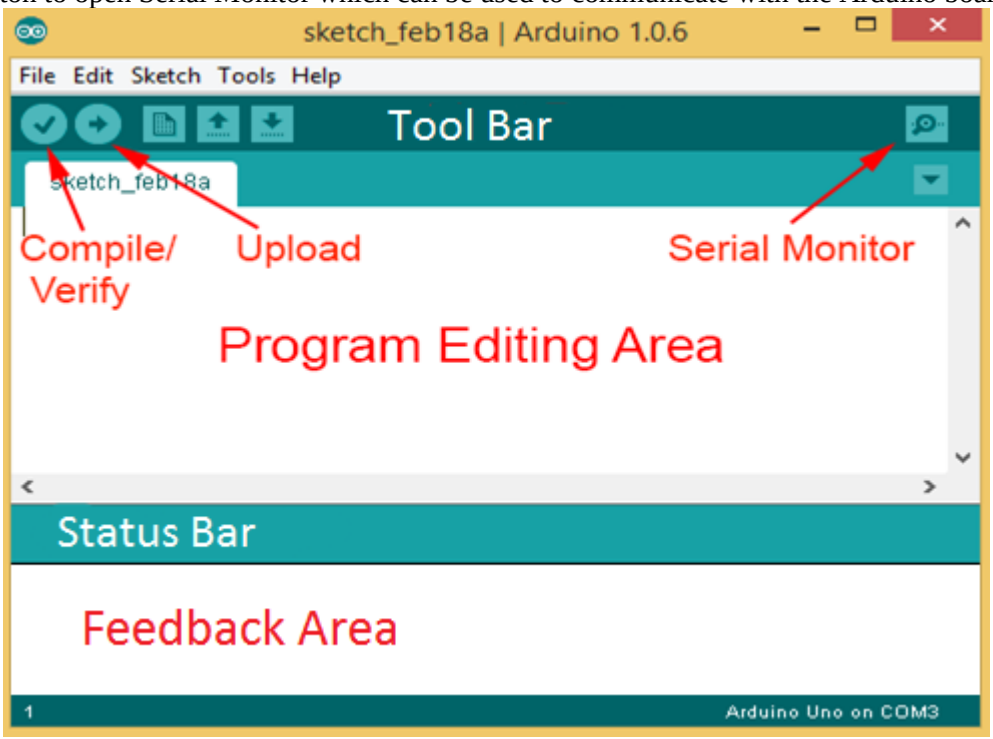



*Fig. 2: Basic Areas and Features of Arduino IDE*

Try to compile this blank file by clicking on the Compile button. You will end up with an error which is indicated at the status bar and the feedback area. Note the error in your code. You will basically come across two errors:

- undefined reference to `setup'
- undefined reference to `loop'

Now write some code in the Program Editing Area to correct the errors raised. Add two functions called "setup" and "loop" as shown below.

```
// the setup routine runs once when you press reset:
void setup() {

}

// the loop routine runs over and over again forever:
void loop() {

}
```

In Arduino programs, one can write comments which will be ignored by the compiler. Any line starting with '//' is a single line comment. Any text in between the '/* and */' is also considered as comments where these can be used to write multiline comments. With the use of the comments we can improve the readability of the code.

Now try compiling the code as you did before. This should give you a successful status as at Status bar; "Done Compiling".

These two functions: "**setup( )**" and "**loop( )**" are two special functions that are a part of every ***Arduino Sketch***. "**Sketch**" is the term used to refer to a unit of code that is uploaded to the Arduino board and run on it. We need to include both these functions "setup( )" and "loop( )" in a sketch in order to make the code compile.

The setup( ) function is called when a sketch starts and it is called once, after each power-up or reset of the Arduino Board. We can use the setup( ) function to do set up tasks like initialize variables, pin modes, start using libraries, etc.
The loop( ) function runs over and over forever as its name indicates. We use the code inside the loop( ) section to actively control the Arduino board.

***Task 3*: Writing a program to blink an LED**

Now modify your program in order to blink the LED that you have connected to the Arduino. The following are some code snippets that you will need in writing the program.

```
pinMode(PIN_Number, OUTPUT);
- Initialize the digital pin, identified by PIN_Number as an output
digitalWrite(PIN_Number, HIGH);
- Make the voltage level high (5V) for the pin, identified by
PIN_Number
digitalWrite(PIN_Number, LOW);
- Make the voltage level low (0V) for the pin, identified by PIN_Number
delay(1000);
- Wait for 1000ms (1 second). The value inside parentheses can be
changed according to the requirement.
int Variable_name = 1;
- Declare an integer variable as Variable_name with value 1.
```

**Hints:**

- You can declare an integer variable with the pin value 12 at the beginning of the program which can be used for the rest of the program.
- Then in the setup( ) function, you can initialize that pin as an output.
- In the loop( ) function, you can make that pin high and wait for 1 second then make it low and wait for another 1 second.
- Since a loop( ) function will run continuously, such program is capable of turning ON and OFF the LED, where it will stay ON for 1 second and OFF for 1 second.
- Making the pin high will supply 5V to the pin which will create a voltage difference across the pins of the LED, and lights it up.
- In between the ON and the OFF, we want enough time for a person to see the change, so we can use the "delay( )" function to tell the Arduino to do nothing for a given period of time.

To help you get started, the program code is given to you below.

```
/*
  Blink
  Turns on an LED on for one second, then off for one second, repeatedly.

  This example code is in the public domain.
 */

// Pin 09 has an LED connected on most Arduino boards.
// give it a name:
int led = 09;

// the setup routine runs once when you press reset:
void setup() {
  // initialize the digital pin as an output.
  pinMode(led, OUTPUT);
}

// the loop routine runs over and over again forever:
void loop() {
  digitalWrite(led, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(1000);               // wait for a second
  digitalWrite(led, LOW);    // turn the LED off by making the voltage LOW
  delay(1000);               // wait for a second
}
```

Compile your code and make sure that there are no compilation errors.

***Task 4*: Uploading the program to the Arduino**

The program you wrote is still in the computer. To run it on the microcontroller of the Arduino board you have to upload (transfer) the code to Arduino board.
If you try uploading the code by clicking on the upload button in the toolbar or by clicking the Upload menu item under File menu (File → Upload), without connecting the Ardiono to PC, it will end up with an error indicating the serial port is not found or serial port already in use. Perhaps you will get a pop up window to input the port number to which the Arduino is connected. This is because you have not connected the Arduino to the PC.
You need to correctly specify the correct port number to which the Arduino is connected, in the Arduino IDE. To identify that, before connecting the Arduino to PC click on Tools and navigate to Serial Port and note down the port numbers specified under that. Then connect the Arduino to the PC with the given USB cable. Again click on Tools and navigate to Serial Port and note the port numbers specified under that. The port number that has not been there before is the port where the Arduino is connected. You need to select that port number by clicking on that.
There are other methods also that you can follow to identify the correct port number. For example, you can go to Device Manager (if you are using Windows OS) and under Ports you can identify the relevant port number.
Now try uploading the code to the Arduino by clicking on the Upload button in the toolbar. If you have no error it will end up by giving you a message; "Done Uploading".
Now observe what happens. The LED connected to the Arduino will blink as expected.

***Task 5*: Playing with blinking LED**

Now try to modify your program for the following cases and observe the outcomes after uploading.

1. Modify the program to blink the LED fast (modify the delay of waiting to 500ms). Save this program with the name L9_P1_E1, in the Lab9 folder.
2. Modify the program so that the LED is ON for 1 second and OFF for 0.5 seconds. Save this program with the name L9_P1_E2, in the Lab9 folder.
3. Modify the program so that the LED will stay ON all the time. Save this program with the name L9_P1_E3, in the Lab9 folder.

Save these three program files (you can use File → SaveAs) as mentioned in a known location, because you have to upload these to Moodle at the end of this lab. Saving a program in Arduino IDE will create a folder with the given name, which includes an ".ino" file (Arduino file extension) inside that.

***Task 6*: Turning the LED ON and OFF responding to a user's input**

Let's try to ON and OFF the LED in response to a user command from the keyboard. If we have entered 1; turn ON the LED and if we have entered 0; turn it OFF. For that task we can use serial communication with the Arduino.
Some additional code snippets that you will need in writing this program are as follows.

```
Serial.begin(9600);
- Open the serial port and set data rate to 9600 bytes per second
Serial.available()
- returns the number of bytes (characters) available for reading from
the serial port
Serial.read();
- read the incoming byte
```

**Hints:**

- In addition to the code that you have written before under Task 3, you can declare an integer variable in order to use to save the incoming bytes.
- In the setup( ) you can open the serial port and set the data rate to 9600 bps (which is commonly used). The data rate, the speed in which the communication happens, is also referred to as the "baud rate".
- In the loop( ) you can check for the availability for the serial data within an IF condition; read it if there is a byte available for reading.
- Then compare the byte read with '1' or '0' and ON or OFF the LED depending on it.
- You will no longer need delays, unlike in task 3 and 4.

Refer the example code given below.

```
int incomingByte = 0;   // for incoming serial data
int led = 10;

void setup() {
       Serial.begin(9600);     // opens serial port, sets data rate to 9600 bps
       // initialize the digital pin as an output.
       pinMode(led, OUTPUT);
}

void loop() {

        // send data only when you receive data:
        if (Serial.available() > 0) {
                // read the incoming byte:
                incomingByte = Serial.read();

                // check what you got:
                if (incomingByte == '1') {
                    digitalWrite(led, HIGH);   // turn the LED on
                                }
                                        else if (incomingByte == '0') {
                    digitalWrite(led, LOW);   // turn the LED off
                                }
       }
}
```

Compile your code and make sure that there are no compilation errors.
Upload the program to the Arduino.
Now open the "Serial Monitor" by clicking the button in tool bar, which can be used to type input from keyboard of PC and then send that as "serial data" from the PC to Arduino. Set its data rate to 9600 (the data rate that we indicated in the program) and see what happens when you send 1s and 0s from that interface. You should be able to turn ON and OFF the LED by sending 1s and 0s from this interface (Fig. 3).

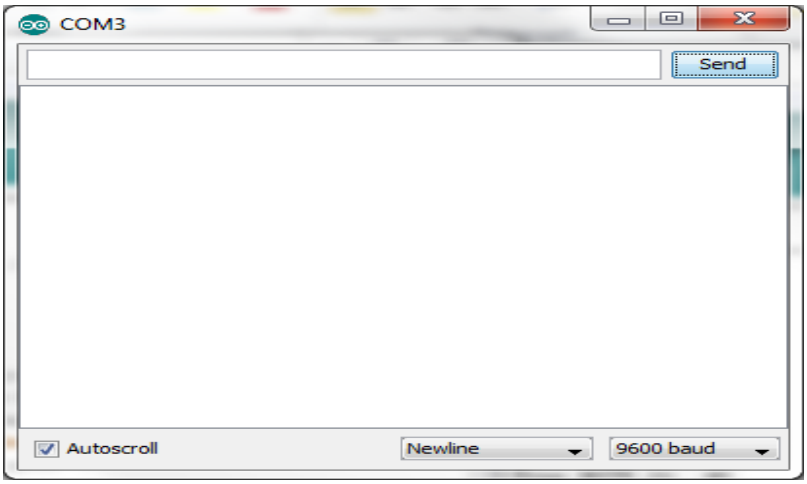



*Figure 3: Arduino Serial Monitor*

Save this program with the name L9_P1_E4.ino, in the Lab9 folder.

## Part 2 – Simple Control Using Arduino Board

**Required Hardware Components**

- Arduino Board
- Computer which is installed with Arduino IDE
- Breadboard
- USB cable (to transfer instructions and data between PC and Arduino board)

- LED – Light Emitting Diode
- LDR – Light Dependent Resistor
- 330Ω Resistor
- 10kΩ Resistor
- Prototype Wires

***Task 7*: Preparing the circuit to control brightness of an LED**

Use the given components and build the circuit that we will use to blink an LED as follows. Connect the positive pin of the LED to pin 10 as indicated. The resistor (330Ω) in series is to limit the current through the LED. Connect the negative pin of the LED to resistor. See Fig. 4.

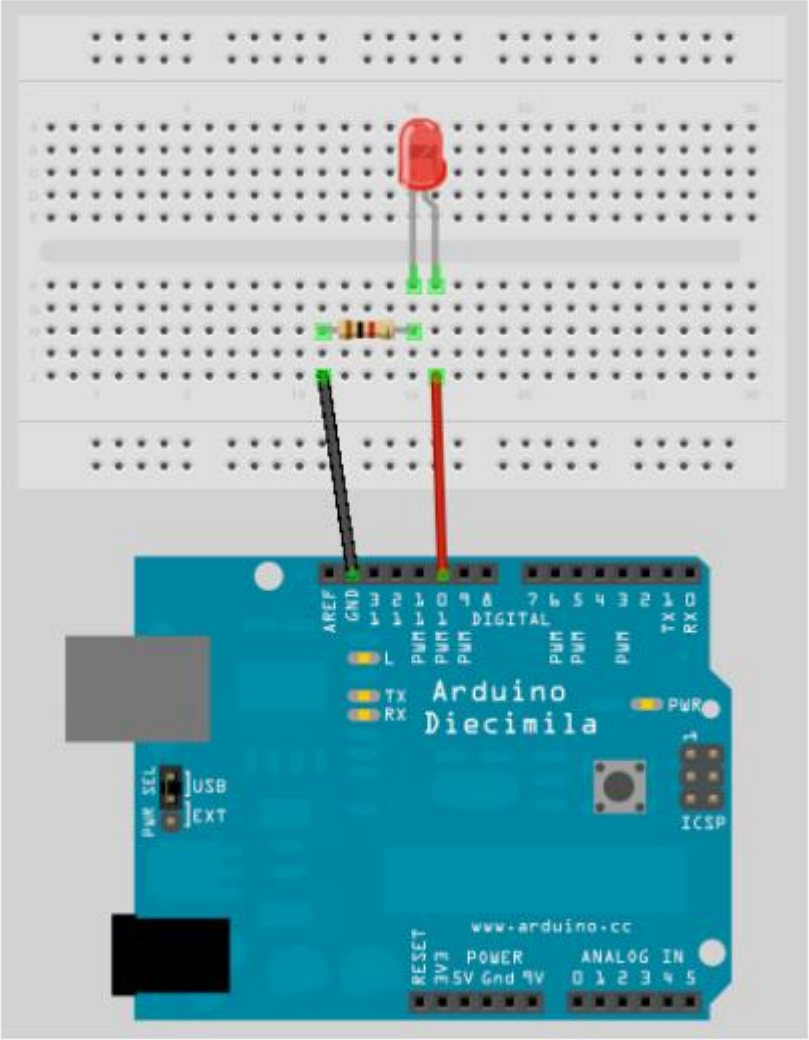



*Figure 4: Connection of the Arduino board for task 7*

Now the circuit for the task 7 (Fig. 4) is ready.

***Task 8*: Writing a program to control the brightness of the LED**

Write a simple program which allows the user to send data from a computer to an Arduino board as input to control the brightness of an LED. Assume that the user sends integers as data, each of which is in the range from 0 to 255. Your program should read the integers sent by the user and use them to set the brightness of the LED.

The following are some code snippets that you will need in writing the program.

```
pinMode(PIN_Number, OUTPUT);
- Initialize the digital pin given by PIN_Number as an output
analogWrite(PIN_Number, Analog_Value);
- Writes an analog value (PWM wave) given by Analog_Value (0 - 255)
to the pin given by PIN_Number
```

```
Serial.begin(9600);
- Open the serial port and set data rate to 9600 bytes per second
Serial.available()
- returns the number of bytes (characters) available for reading from
the serial port
Serial.parseInt();
- returns the first valid integer number from the serial buffer
```

**Hint:**

On the Arduino board that is given for this lab, the "analogWrite( )" function works only on pins 3, 5, 6, 9, 10 and 11. These pins are also called PWM pins. Pulse Width Modulation, or PWM, is a technique for getting analog results with digital means. Digital control is used to create a square wave, a signal switched between on and off. This on-off pattern can simulate voltages in between full on (5 Volts) and off (0 Volts) by changing the portion of the time the signal stays high versus the time that the signal stays low. The duration of "on time" is called the pulse width. A call to analogWrite( ) is on a scale of 0 - 255, such that analogWrite(255) requests a 100% duty cycle (always on), and analogWrite(127) is a 50% duty cycle (on half the time) for example (Fig. 5).

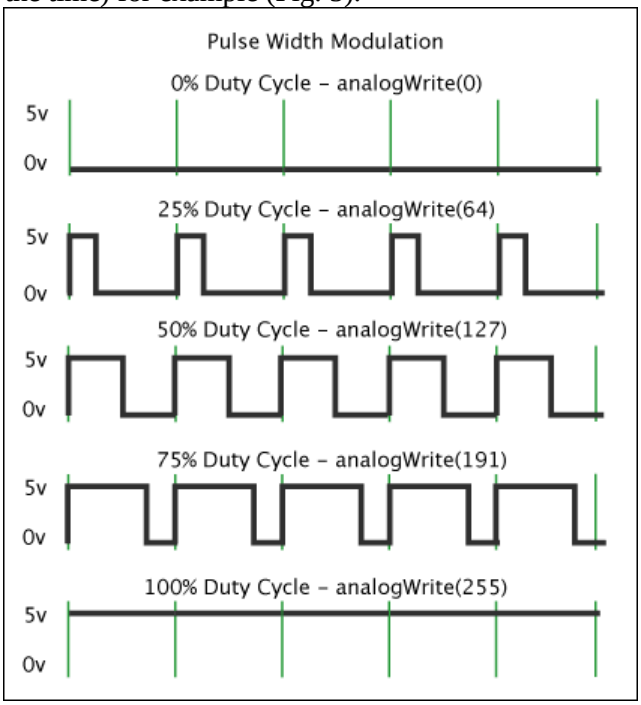



Figure 5: PWM for various input values

Compile your code and make sure that there are no compilation errors.
Upload the program to the Arduino.
Open the Serial Monitor and set its data rate to the same data rate that you have indicated in the program and see what happens when you send integers varying from 0 to 255 from that interface.
Show your outcome to the instructor before proceeding to the next step.
Comment your code to explain what you have done.
Save this program with the name Lab9_P2_E1, in the Lab9 folder.

***Task 9*: Preparing the circuit to get an input from LDR (Light Dependent Resistor)**

Connect the LDR and the resistor (10kΩ) to pin A0 in order to create a voltage divider, which will be sensitive to the amount of light incident on the LDR. The 10kΩ Resistor should be connected between pin A0 and 5V POWER input. The LDR should be connected between pin A0 and ground pin (GND). Now the circuit for the task 9 onwards of the lab is ready.

***Task 10*: Writing a program to read the input from the LDR**

Write a simple program to read an analog value from the pin that the LDR is connected (pin A0) and print it to the serial port continuously.

The following are some additional code snippets that you will need in writing the program.

```
analogRead(PIN_Number);
- Reads the value from the specified analog pin. This returns a value
in the range 0-1023. 0 maps to 0V and 1023 maps to 5V
```

```
Serial.println(value);
- Prints value (value can be any data type) to the serial port as human-readable ASCII text followed by a carriage return character and a newline character
```

Compile your code and make sure that there are no compilation errors.
Upload the program to the Arduino.
Open the Serial Monitor and observe the values that are being printed. Make note of the value range when the LDR is fully covered by hand and when the LDR is not covered at all. Check whether there is at least a gap of 500 between the two ranges. If not you might want to change the lighting conditions.
Now referring those values obtained, figure out a threshold value (most likely the mid value of the above gap) which will clearly separate the LDR inputs when the LDR is covered from the LDR inputs when it is not covered. Note down this threshold value to be used for the rest of the lab.
Comment your code to explain what you have done.
Save this program with the name Lab9_P2_E2, in the Lab9 folder.

***Task 11*: Writing a program to change blinking behavior depending on the light intensity**

Write a simple program to blink an LED with different behaviour depending on the light intensity.
**When the LDR is covered:** Blink the LED with the pattern: ON for 0.5 seconds and OFF for 0.5 seconds
**When the LDR is not covered:** Blink the LED with the pattern: ON for 1 second and OFF for 0.5 seconds

Compile your code and make sure that there are no compilation errors.

Upload the program to the Arduino.

Show your outcome to the instructor before proceeding to the next step.

Comment your code to explain what you have done.

Save this program with the name Lab9_P2_E3, in the Lab9 folder.

***Task 12*: Uploading the work done to the LearnOrg-Moodle system (https://online.mrt.ac.lk/)**

Upload the Lab9 folder containing all program files that you have saved under Part 1 and Part 2 as a single zip file named “Lab9_ IndexNo.zip” on to the relevant location of the learnOrg-Moodle.

***Task 13*: Playing with Arduino**

What you have done up to now are simple and basic tasks that you can do using an Arduino. There are several examples that come along with Arduino software that you can access from Files > Examples. If you have completed all the tasks up to now and have time remaining in the lab, try them or try out your own programs for the rest of the time in the lab session.

# Lab 10b: Embedded Programming for Simple Applications

**Objective**
After successfully completing this lab session, you should be able to:

- Develop a simple real world application using Arduino board.
- Use logics and loops in an embedded system program.

**Prerequisites:** Students must have successfully completed the Lab 9 individually.

**Remarks**

- This lab will be **evaluated** and will be done individually.
- All the source codes should be saved in a folder named **Lab10** in the **home** directory.
- The files should be uploaded as a zip **Lab10_IndexNo.zip** to learnOrg-Moodle.
- Be careful when wiring; do not connect power and ground lines together at any time.
- If you need any guidance with this lab ask for the support from an instructor.
- Before leaving the lab, show your outcomes to the instructor.

## Part 1 – Simple Traffic Lights

**Required Hardware Components**

- Arduino board
- Computer installed with Arduino IDE
- USB cable
- Breadboard and Prototype Wires
- 3 LEDs – Light Emitting Diodes (Red, Yellow and Green)
- Three 330Ω Resistors

***Task 1*: Preparing the circuit**
Use the given components and build the circuit as shown in Fig.1.

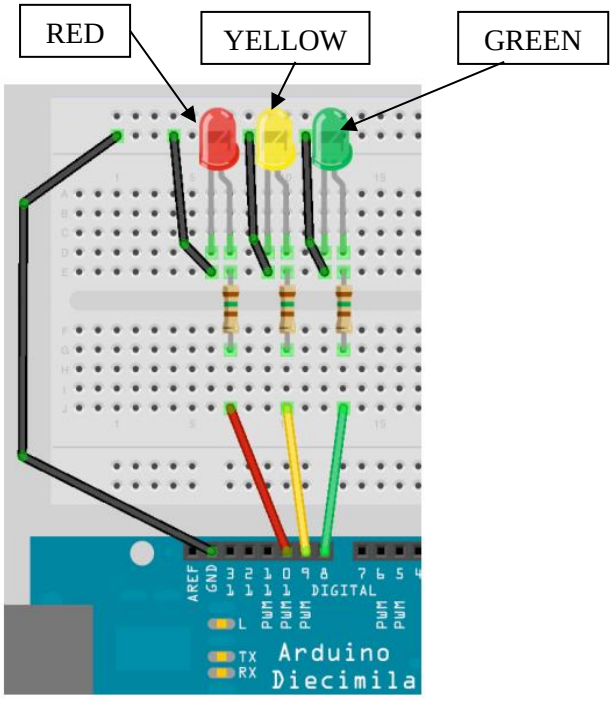



*Figure 1: Connections on the Arduino board*

***Task 2*: Writing a program to control traffic lights.**
Write a program which will change the traffic light from red to yellow, yellow to green, green to yellow and yellow to red, continuously. At a given time both red and green or both yellow and green lights

should not be turned ON together. The red light or green light should be ON for 5 seconds before changing. The yellow light should be ON for 2 seconds.
The following are some code snippets that you will need in writing the program.

```
pinMode(PIN_Number, OUTPUT);
- Initialize the digital pin given by PIN_Number as an output
digitalWrite(PIN_Number, Digital_Value);
- Writes an digital value given by Digital_Value (HIGH/LOW) to the pin
delay(TIME)
-  To keep the state of traffic light for a period of time.
```

Compile your code and make sure that there are no compilation errors.
Upload the program to the Arduino.
Show your outcome to the instructor before proceeding to the next step.
Comment your code to explain what you have done.
Save this program with the name Lab10_E1, in the Lab10 folder.

## Part 2 – Interactive Traffic Lights

**Required Hardware Components**

- Arduino board
- Computer which is installed with Arduino IDE
- USB cable
- Breadboard
- 2 Red LEDs, 2 Green LEDs, 1 Yellow LED
- Five 330Ω Resistors
- 1 kΩ Resistor for button
- Push button switch
- Prototype wires

***Task 3*: Preparing the circuit**

Use the given components and build the circuit as shown in Fig.2.

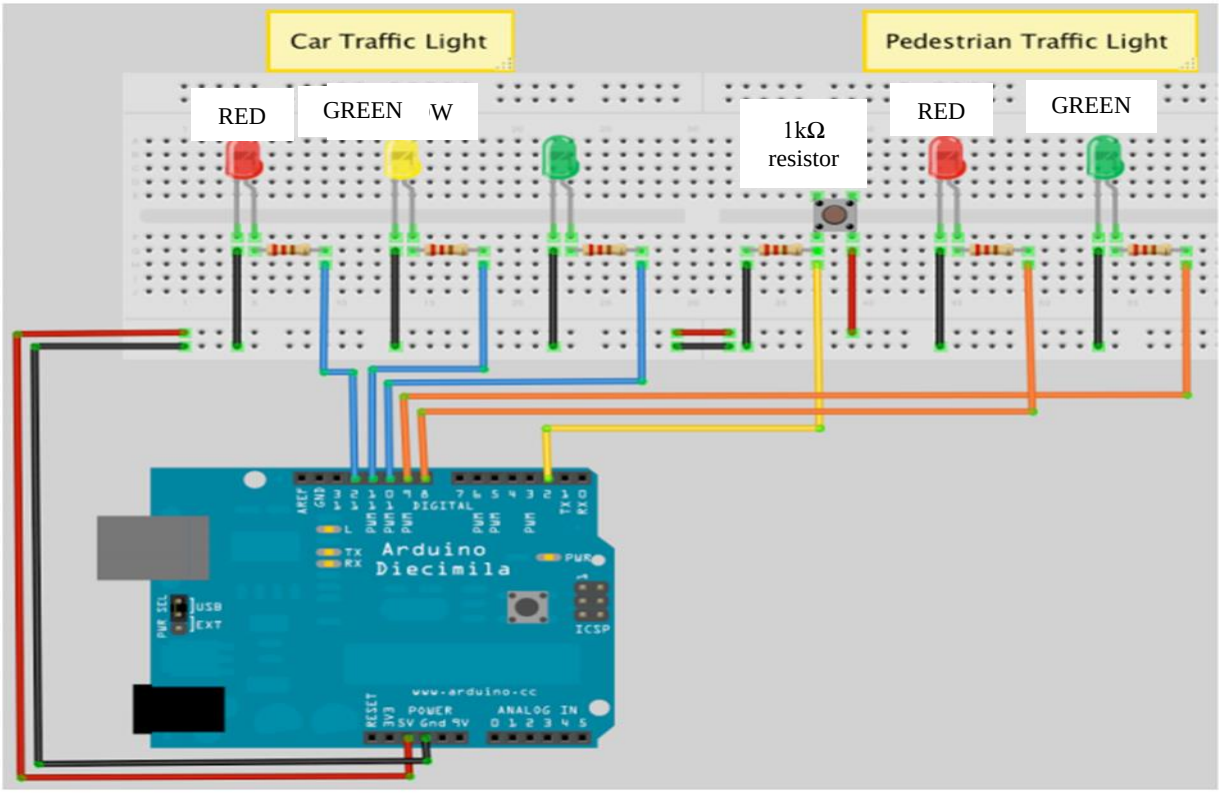



*Figure 2: Circuit for the interactive traffic lights*

***Task 4*: Writing a program to implement an interactive traffic light system.**

Extend the Part 1 of the lab to include a set of pedestrian lights where a pedestrian pushes a button to request to cross the road. The Arduino will react when the button is pressed by changing the state of the lights to make the cars stop and allow the pedestrian to cross safely.

The pedestrian crossing green light should be visible for 5 seconds. If there are no pedestrian requests the traffic light for vehicles will be green continuously.

Between two pedestrian green lights vehicles should be allowed at least for 5 seconds. When the pedestrian switch is pressed, the program should check whether at least 5 seconds have gone by since the last time the lights were changed (to allow traffic to get moving).

**Hints:**

- To reduce the complexity of the program use a function changeLights( ) to change the traffic light colours.
- Use in-built function `millis()` to get the number of milliseconds since the Arduino board began running the current program .
- Get the last time the pedestrian lights changed to red (`lastChangeTime`) and compare the difference of that value to the value returned by millis( ) when button is pressed again. You can use this to ensure the vehicles run for at least 5 seconds before pedestrian lights turn green.
- The following is a code snippet that may be useful in writing your program.

```
void loop() {
  //Read the status of the button
  int state = digitalRead(button);
  /* check if button is pressed and it is
  over 5 seconds since last button press */
  if (state == HIGH && (millis() - lastChangeTime) > 5000) {
    // Call the function to change the lights
    changeLights();
  }
}
```

Compile your code and make sure that there are no compilation errors.

Upload the program to the Arduino.

Show your outcome to the instructor before proceeding to the next step.

Comment your code to explain what you have done.
Save this program with the name Lab10_E2, in the Lab10 folder.

***Task 5*: Upload the work done to LearnOrg-Moodle system (https://online.mrt.ac.lk/)**

Upload the Lab10 folder containing 2 program files that you have saved under Part 1 and Part 2 as a single zip file named "Lab10_ IndexNo.zip" on to the leanrOrg-Moodle system.