

FIT2022 Tutorial 1

1. Objectives

In this first pair of tutorial/laboratory sessions, we focus upon objective 6. Operating systems are **programmed systems**, and we need to do some programming to understand what goes on inside the operating system. Most of this tutorial is directed at learning some basic Python coding, since Python is the language in which the laboratory exercises are to be undertaken.

Students have asked in the past: "Why do we have to learn another programming language?" . Unfortunately (or fortunately, depending upon your point of view), there are many, many programming languages, characterized only by one thing: they were all designed to address programming tasks. Some focus upon efficiency, some upon clarity, some upon particular application domains, and some upon teaching. Python is one of the so-called *scripting* languages, but it is also *object-oriented*, so you can see that sometimes designers have more than one target in mind when they design a language. Computer Scientists and Software Engineers just cannot get away with only knowing one or two programming languages, and even more importantly, they need to know languages from a range of paradigms.

The other serious contender for use in a computer systems unit is the C Programming Language. While most of you will have learnt Java (and there are some significant similarities between the syntax of Java and C), C is a difficult language to learn and use effectively. You just don't have time in a single semester unit to learn about C **and** operating systems. For that reason, we have elected to use Python. As we suggest above, you'll probably have to learn C at some stage in your career, but now is not that time.

Why Python? Well, it is a modern, object-oriented language, with clear syntax (one of its unusual features is that it uses indentation to specify nested structures), and excellent error messages. All of the modern programming artifacts are in it, and it comes with an excellent library that allows the programmer to quickly and effectively develop programs. Most importantly for this unit, it has an `exec` function, which allows us to do the 'magic' of executing the programs that we write. More of this anon.

2. Outcomes from this Tutorial

This tutorial is addressing objective 6. At the end of this tutorial, you should:

1. Have a basic understanding of the syntax of Python;
2. Be able to write a few simple programs in Python; and
3. Appreciate the value of rapid prototyping in a scripting language.

3. Setting Yourself Up with Python

The lab sessions are based upon using Linux boxes, and these have all been set up with Python systems. Many of you will have personal computers, and will want to run your programs on those systems. If you have Linux, then you can install the same system as is on the lab machines. But Python is fairly consistent across platforms, anyway. Instructions on installing Python are on the Resources Page .

4. Introduction to Python

Most of this tutorial is dedicated to discussing Python. It is suggested that you work through the Informal Introduction to Python that is in the tutorial document on Python. But feel free to explore your own teaching resources.

Exercise 1

Write a sequence of Python statements to assign your given name and family name to the two variables `given` and `family` respectively, and then to print out your complete name in a single line.

Exercise 2

Write a sequence of Python statements to build a list, containing your given name, a space character, and your family names. Now alter the middle element of the list to contain your middle name (make one up if you don't have a middle name!)

Exercise 3

write a Python function `addname` with two parameters, `given` and `family`, that adds a full name *given family* to a list of names of people, called `names`. This program won't work if you try it out - you need to declare the list of names to be `global`, see later.

5. The Mutual Update Problem

5.1 Introduction

- + Context is **Process Synchronization**: How do we get two independent processes to synchronize their operations in a reliable, correct, fashion?
- + Why is this difficult? To answer this question, first look at the **Mutual Update Problem**.
- + *Learning Objective*: To understand the Mutual Update Problem, how it arises, and the need for **atomic operations** in its solution.

5.2 Background to the Mutual Update Problem

- + Concurrent access to shared data may result in data inconsistency.
 - + Example: Airline booking system
- + Can characterize problem thus:
 1. Suppose we have two processes A and B that both want to update some shared variable `n`

2. Each does $n := n + 1$, which involves reading the value of n , adding 1 to it, and then storing the result back to n . Call these operations R, I, S for read, increment, store.
3. Process A performs operations R_A, I_A, S_A in order; while process B does R_B, I_B, S_B , also in order
4. What happens when the speed of execution of R,I,S varies, and process A overlaps with process B?

5.3 The Mutual Update Problem

- + The problem arises because while the order of each R_A, I_A, S_A ; and R_B, I_B, S_B is guaranteed by program semantics *within a single process*, it is not guaranteed *between concurrent processes*. This is called a *race condition*, since the actual result depends upon the interleaving of the instructions
- + Consider the following scenarios, assuming $n=3$:

Time	Process A	Process B	Time	Process A	Process B
0	$R_A(=3)$		0	$R_A(=3)$	
1		$R_B(=3)$	1	$I_A(=4)$	
2	$I_A(=4)$		2		$R_B(=3)$
3		$I_B(=4)$	3	$S_A(=4)$	
4	$S_A(=4)$		4		$I_B(=4)$
5		$S_B(=4)$	5		$S_B(=4)$

- + In both cases, the value of n ends up as 4, when it should be 5

5.4 Solution to the Mutual Update Problem

- + How do we solve the Mutual Update Problem?
- + Note that the correct behaviour is guaranteed if we make sure each R,I,S sequence is *not overlapped* with the other:

Time	Process A	Process B	Time	Process A	Process B
0	$R_A(=3)$		0		$R_B(=3)$
1	$I_A(=4)$		1		$I_B(=4)$
2	$S_A(=4)$		2		$S_B(=4)$
3		$R_B(=4)$	3	$R_A(=4)$	
4		$I_B(=5)$	4	$I_A(=5)$	
5		$S_B(=5)$	5	$S_A(=5)$	

- + The order of whether A goes first or B goes first does not matter, as long as they *do not overlap*

5.5 Conclusions

- + The read, increment and store operations must be performed as an **atomic operation**.
- + Sections of code that access shared variables (such as n) are known as **critical regions**
- + No two processes can be executing in critical regions simultaneously. This ensures the operations in the critical region are performed atomically.
- + This is the basis for understanding much of **concurrent process synchronization**, and **database locking protocols** (to follow).