CITS1401

Daniel Paparo

1 Lecture 1

- CITS1401 is computer based problem solving
- Uses python 3

Assesments include a midsemester test in week 9, two programming projects in week 8 and week 13, and a final examination.

2 Lecture 2

Programming is essentially following recipes - Things we need to do to achieve an end.

Program - Essentially just a list of instructions to be executed by the computer

Computer Science - What sort of computations can we describe

We can design algorithms to do things - A set of steps to achieve a desired result.

Design - What is computable. Not being able to design an algorithm does not mean it is unsolvable.

Analysis - The process of examining algorithms and problems mathematically.

Experimentation - Implementing a stystem and then studying its behaviour under different conditions.

2.1 Hardware Basics

CPU - Carries out simple computations.

RAM - Fast volatile memory.

Secondary memory - Permanent storage such as HDD, Optical, Magnetic and SSD.

IO Devices are the way in which we interface with the computer and the way it interfaces with us.

2.1.1 Fetch Execute Cycle

1. Load program into the main memory
2. Fetch the next instruction from memory
3. Decode the instruction to see what it represents (Fetch operands, as required)
4. Carry out the appropriate instructions

2.2 Programming languages

Natural languages cannot precisely describe an algorithm, and therefore we need programming languages to express algorithms in a precise way.

Syntax - Every structure in programming languages follow a precise form.

Semantics - Every structure in the programming language has a precise meaning.

High-level programming languages - Designed to be understood by humans but are typically run much slower

Low-level programming languages - Much more difficult to be read by humans but will run much faster.

Compilers- Compilers will convert programs written in high level languages to machine language in one go.

Interpreters - Convert to machine language one instruction at a time (at run time).

3 Lecture 3

Function: Body of code wich is grouped by a like purpose

- Indentation does matter in python
- A function must be invoked

Use functions for:

- Defining once, and then using many times
  - Dont Repeat yourself (DRY)
- Aides problem decomposition
  - Makes big problems smaller
- Defining code as functions allows for independednt testing and validation of code

Unit testing: Testing independent functions, and checking they work by themselves
Module File: A plain text document with function definitions (.py)
Programming enviroment: IDE

- Indenting in python shows scope
- Indenting is extremely important in python! It shows the block structure!

A small example below:

```
In [ ]: # A simple program illustrating chaotic behaviour
        def main():
            print("This program illustrates a chaotic function")
            x = float(input("Enter a number between 0 and 1:"))
            for i in range(10):
                x=3.9*x*(1-x)
                print(x)
        main()
```

```
This program illustrates a chaotic function
```

Functions must be invoked - defining the function only allows it to be invoked.

3.1 Importing modules

We need to import modules into the python interpreter, and is done in the following:

```
import chaos
import math
```

Notice the lack of .py attatched on the end. When python imports a module, it executes each line. When calling a function from a library we must use the following format:

$$moduleFileName.functionName()$$

Such as: math.sqrt(2) chaos.main()
And this will only work once it has been imported.

4 Lecture 4

4.1 Software development

First we need to figure out exactly what the problem is. Understanding the users view point is one of the best ways to find out the program requirements

To determine the specifications we need to figure out: how do the outputs relate to the inputs?

4.2 Creating a design

Formulate the overall structure of the program - Without coding yet, this is called the module level. You can either chose a program or build one which meets the specification ## Implementation of the design Translating the design into a programming language!

4.3 Teset/Debug the program

Potentially the most important step!

Syntax errors: These are highlighted by the interpreter and need to be fixed before the program will work at all!

Logical error: When the sequence of instructions are legal, and the program will run, but will not compute the intended function.

Debugging: Locating and fixing any errors(bugs)

- Users will find the bugs if they are present
- The goal is to find all errors that might break your program!

Antibugging: Code in the program to catch potential bugs from outside data

4.4 Maintaining the program

Continuing development after the program has been released - including respdonging to users changing needs. This is referred to as the software life cycle.

4.5 Elements of programs

4.5.1 Naming

Names are given to variables, functions, modules etc. These names are called identifiers, and are case sensetive.

4.6 Elements

Some identifiers are part of python itself - these are known as reserved words and are not available for you to use as a name for a variable - these reserved words are vailable online.

5 Lecture 5

There are always two audiences for the source code: The computer, and other humans. As such we must use whitespace whenever possible to make the code as readible as possible - but also used to show indentation for scope.

5.1 Output statements

Essentially to talk to the outside world:

- Print statements can print any number of expressions
- print() will first go inside the brackets and evaluate that expression first

5.2 Input

```
z = input("type a value")
```

To recieve input from a user or from the computer/ another computer. By using input with another operator such as 'int()', if the input is not an integer an error will be thrown, otherwise the string input will be converted to an integer.

5.3 Variables

$$< variable >=< expression >$$

Everything in python is pointing to some other object.

5.3.1 Simultaneous assignments

Several values can be calculated at the same time:

```
<var1>,<var2>=<var2>,<var1>
```

This example will swap the variables.

5.3.2 Increment assignment

```
x=x+n
x=x-n
x=x/n
x=x*n
```

Can be simplified as:

```
x+=n
x-=n
x/=n
x*=n
```

5.4 Loops

5.4.1 Definite loops

A defenite loop is one which is bonded by some range, or number of iterations:

```
for <var> in <range>:
    <statements>
```

Loop index: The counter of the number of iterations the loop has done
As a loop oalters the flow of the program it is often reffered to as a control structure.

6 Lecture 6

Computer Science: Finding answers through design, experimentation and analysis. Specifically it is about designing, analysing and evaluating algorithms

6.1 Numeric types

Integers: Whole numbers
Floating point numbers: Real numbers (decimals)
Datatype: Determines what values an intem can have and what operations can be performed apon it

```
In [ ]: #Does a floating point division
        print(10/3)
        print(3.0/3.0)

        #Does an integer division
        print(10//3)
```

6.2 Type conversion

We can explicitly switch between types in the following

```
In [ ]: x = int(6.8)
        y = round(6.8)

        print(x, y)
```

7 Finding a datatype

Using the `type` function we can find the datatype of a variable

```
In [ ]: print(type(4.2))
        print(type(2))
        print(type("Daniel"))
```

7.1 Scientific notation

Pythons floating point numbers are expressed in scientific notation (`3e-2` would be 3×10^{-2}). The floating point numbers can range from -10^{308} to 10^{308} with a typical precision of 16 decimal places.

7.2 The math library

To import the math library simply:

```
In [ ]: import math
```

To access a function from this freshly imported module all we need to do is:

```
In [ ]: print(math.sqrt(4))
```

The "dot notation" tells python to use the `sqrt` function from the `math` module.

8 Lecture 7

8.1 Accumulator algorithms

This is moving towards the idea of software paterns.
Design patterns: Reccuring sets of reusable sets of instructions
The accumulator patter:

1. Initialize accumulator variable
2. Perform some computation
3. Update accumulator variable
4. Loop until final result is found
5. Output final result

8.2 Integer problems

Integers are expressed by a 64 bit integer (on most modern computers). Floats are only approximate. Python can handle larger intergers than 2^{64} by simply adding more integers to the end (behind the scenes). The problems with integers are relatively easy to fix and are mostly dealt with behind the scenes

8.3 Float problems

Floats are always an approximation. Very small and very large values present a bad situation

For example:

```
In [ ]: x = 1.0e308
        x *= 100
        print("x =", x)

        y=1e-308
        y/=10000
        print("y = ", y)
```

This is an example of overflow and underflow (even though underflow in this example does not appear to be working). These can create huge problems in a program as it will not cause any error, but instead will mess things up later in the programs life.

Logical errors are much more difficult to detect than syntax errors!

8.4 Decision structures

Decision structures are used to change the sequential execution flow of the program ### Simple if statements

```
if<condition>:
    <statement>
```

In this example the `<condition>` is a boolean statement which is evaluated to be either `True` or `False`. If it is true, then the body of the if statement, `<statement>` will be executed.

8.4.1 A boolean expression

$$< expression >< relationaloperator >< expression >$$

The relational operators are as follows:

- `a==b` - a and b are equal
- `a!=b` - a and b are not equal
- `<>` - returns true if the two sides are not equal
- `>` - if left is greater than right
- `<` - if right is greater than left
- `=>=` - if left is greater than or equal to right
- `<=` - if right is greater than or equal to left

Logical boolean operators are as follows: * `and` - left and right are true * `or` - left or right or both are true * `not` - if expression is true, it becomes false, and opposite

9 Lecture 8

9.1 Object oriented python

An object is an active datatype which combines data and operations:

- Objects know stuff (contain data)
- objects can do stuff (have operations)

Objects interact with eachother by sending messages.
Example: A student object will have the following attributes:

- a name
- Phone number
- Residential address
- Units being taken

And can have the following operations put on them:

- Respond to requests for data such as: being asked for their name or phone number

A course object will have:

- An instructor
- List of students enrolled
- Pre-requisite courses
- When and where the classes meet

Some operations of the course object would be:

- `addstudent()`
- `delstudent()`
- `changeRoom()`

An object can, itself, contain one or more objects from within itself; for example the course object will have many student objects inside.

9.2 Graphics programming in python

To use the graphics library we must first import it, then set it up as a new window:

```
In [ ]: import graphics #Imports the library
        win = graphics.GraphWin() #Creates the new window object

        win.close() #Closes the window
```

This code generates a new window object named "graphics window"

9.3 Importing library functions

To avoid using "dot notation" after importing a module (which can often be tedious), many python programmers will import in the following way:

```
In [ ]: from graphics import *
        win = GraphWin()

        win.close()
```

This imports all function from `graphics` and also allows for the functions to be called without dot notation. This can be bad for debugging however (as it will make it difficult to know where each of the functions came from), and therfore the following option is preferable:

```
In [ ]: import graphics as gx
        win = gx.GraphWin()

        win.close()
```

9.4 Graphics and objects

When called, `GraphWin()` generates a new object which is assigned the variable `win` in our program. We can manipulate this object through that variable, such as with `win.close()`

9.5 Class-instance-object model

Class: A template to create an object
Instance: A unique copy of the class
Object: An instance of the class

10 Lecture 9

Contructor: an instance of the object is constructed
Instance variables: Variables which are inside an instance of an object

10.1 String data type

A python class which can accept words, sentences and characters. Strings can mostly be treated in the same way as a list or array in python, as the string is essentially a list of characters.

```
In [ ]: s = "Daniel is " #Declaring a new string
        word = "awesome" #Declaring a new string

        sentence = s + word #Concatenating a string

        print(sentence)

        print(sentence[3]) #same as with lists, an individual index can be accessed

        print(sentence[0:6]) #accessing from index 0 and index 6
```

10.1.1 String inputs

we can ask the user for input, which is given as a string - if we wish to do numerical operations with the string we must first convert to an integer

```
In [ ]: userinput = input("What is your name? ") #Asks the user for an input string

        print("Your name is a string: " + userinput) #Concatenates the strings

        print(userinput * 4) #Prints the string 4 times

        print(len(userinput)) #Provides the length of the string

        print(userinput.upper()) #Converts the string to all uppercase

        print(userinput.lower()) #Converts to all lower case
```

11 Lecture 10

11.0.1 String representations

The requirement for standardisation in formatting and coding was realised early in computers. ASCII (american standard code for information interchange) uses 127 characters using 8-bit codes. Python supports unicode (100000+ characters) using variable width words, but cannot be written in unicode - only ascii.

Every letter in the alphabet is in fact a number, we can find this in the following functions:

```
In [ ]: print(ord("A"))
        print(ord("a"))
        print(chr(97))
        print(chr(65))
```

Accumulator pattern: Starting with something empty (such as a string) and then adding to it.

```
In [ ]: csv = "Daniel,John,,Paparo"

        print(csv.split(','))
```

12 Lecture 11

12.1 Cryptography

Cryptography is essentially modifying some text into some form which unwanted listeners cant understand, but a wanted listener can do some operation to get back to the original form

12.1.1 Substitution Cipher

Changing the letters of a string to that of some cipher alphabet - this is relatively easy to crack using different methods such as comparing the frequencies of letters.

12.1.2 Encoding to encryption

Private key: Everyone who is involved in the encryption/decryption needs to have some key
Symmetric-key Crpyography: Sender and reciever have the same key, this is not very safe

12.1.3 Public key or Asymmetric cryptography

There are seperate keys for encrypting and decrypting the message. There is a public key which is publicly available to encrypt, and a priave key to decrypt. Anyone with the public key can encrypt, but only someone with the private key can decrypt.

12.2 Lists

Lists look and work much like strings, with the main difference being that a list can contain anything, not just characters.

```
In [ ]: print([1,2]+[3,4]) #Concatenation
```

Something special about lists is that, unlike a string, a list is mutable, and as such it can be changed after its creation.

```
In [ ]: myList = [1,2,3,4]
        myList.append(5) #Appending the number 5 to the end of the strings

        print(myList)
```

13 Lecture 12

13.1 String formatting

The template contains specifier slots with descriptions:

$$< index >:< format - specifier >$$

where the format specifier has:

$$< width > . < precision >< type >$$

For example:

```
In [ ]: print("Compare {0} and {0:0.20f}".format(3.14))
```

The above example shows the approximate nature of floating point numbers. An example of tihs is in the following:

```
In [ ]: def multiplication_table():
            for i in range(11):
                for j in range(11):
                    print("{0:0d} x {1:0d} = {2:0}".format(i,j,i*j))

        # multiplication_table()
```

13.1.1 Multiline strings

The new line character `\n` is used in a string to denote a new line. The use of `"""` is available for having strings spread accross multiple lines. The tab character `\t` is available to add tabs to the file.

13.2 Files

A text file is essentially a multi-line string. We want to use files as it allows for the data to be kept outside of the program, not storing within the program itself, this means the files dont disappear when the program closes.

13.2.1 File processing

The process of opening a file involves associating a file on disk with an object in memory. We can minipulate the file by manipulating this object:

- Read from the pipe
- Write to the file

When a file is opened, the content is read into ram (either line by line or all at once).
To open the file:

$$< filevar >= open(< name >, < mode >)$$

The methods:

- `<file>.read()` - Returns the entire contents of the file as a single string.
- `<file>.readline()` - Returns a single line as a string

Batch mode processing: is where program inputs and outputs are done through files (the program is not designed to be interactive).

14 Lecture 13

14.1 Exception handling

Exceptions are there to deal with potential errors - These errors may be unlikely, but all the same will break the problem. When the program detects an error condition it will rais an error. Programming languages use a mechanism called exception handling to solve this issue.

An exception is essentially helping the program "fail gracefully". Python deals with exceptopns in the following:

```
In [ ]: try:
            # This body is tried
            int("Daniel")
        except: #this can also be except <error-type>:
            print("ERROR!")
            #Will be run in the event of an exception being caught
```

The try except pattern is comparible to the `if else` structure.

Note that: if there is an exception in the exception handler, then it must be caught or the program will still fail. Also note that `if` statements should be preffered over known errors, `except` should be used for errors which are more abstract but can still occur. For example: `if` would be used to guarentee that a number is between 1 and 100, but an `except` would be used to detect if the input is not a number.

14.2 The main function

The real main program in python is `__main__`, this is used as the main entry point of the function, and will be executed as soon as the program begins.

15 Lecture 14

Most list operations do not return a value - they change the contents of the list in some way. ## Tuples A tuple is a list with round brackets. While a list is mutable, the tuple is not. To turn a list into a tuple we simply use the `tuple()` function.

15.1 Dictionaries

After lists, the dictionaries are the most widely used collection/compound data type. A list is sequential - a dictionary has a register to lookup certain elements. |

```
In [ ]: names = {"Daniel": "20", "Joseph":"32"}

        print(names["Daniel"])
```

Dictionaries are mutable, and as such can be edited after their creation.

`strip()`: Removes white space and terminating characters at beggining and end of a line.

15.1.1 Dictionary operations

`list.get()`: An elegant way to check a dictionary for a value which may not be known.

```
In [ ]: print(names.get("Daniel", "Unknown!"))
        print(names.get("Aaron", "Unknown!"))
```

16 Lecture 15

16.1 Functions

Functions reduce repitition - problem decomposition. A function is almost like a sub-program. When a function is compiled it does not run - it just gets turned into a ready to go function.

Refactoring: putting repeated code into functions.

Variable Scope: Variables are only visable at their block level - variables in a function are only in scope within their function.

When a function is called:

1. The calling program suspends execution at the point of the call (gets pushed to the stack)

2. The formal parameters of the function get assigned the value supploed by the actual parameters in the call.
3. The body of the function is executed.
4. Control returns to the point just after where the function was called (gets popped from the stack).

16.2 Returning from a function

Once the function has completed its process, it will return its value using the `return` clause.

- All functions return a value even if there is no return statement
- Functions without a return hand back a special object, denoted `None`.

Pass parameters by value: Values will be returned, but the original value is left unchanged.
Pass parameters by reference: Values will be modified in their memory location, no need to return.

17 Lecture 16

Lists are a mutable datatype - When this has been passed it can be modified in place. The container (list) will not be changed, but the contents can be (values inside it). This means that we can pass a list by reference.

```
In [2]: def modify_value(array):
            array[0] = 20  #This would normally be out of scope for an integer\

        def main():
            array = [0,0,0,0] # This will be modified in place
            modify_value(array)
            print(array)  # With an integer this wont be the modified value, but the original

        main()
```

```
[20, 0, 0, 0]
```

17.1 Default parameters in function

A default value can be set for when function parameters may potentially not be set:

```
In [3]: def printfunc(words = "Hello Word!"):  # Adds a default value, if the user does not inpu
            print(words)

        printfunc("Test!")  # Does not use the default value
        printfunc()  # Will use the default value
```

```
Test!
Hello Word!
```

This is important for minimising the amount of redundant information passed accross.

A lot of what we are creating is in order to make our programs more modular - we want to be able to reuse our data.

18 Lecture 17

18.1 Loops

Infinite Loop: A loop in which the condition can never allow for termination.

```
In [ ]: def infiniteLoop():
            while True:
                print("Hello!")

        # infiniteLoop()
```

Nested Loops: Allow for loops within other loops - This can help deal with multiple problems simultaneously.

18.1.1 Designing nested loops

- Design the outer loop without knowing what the inner does
- Design whats going on insie, while ignoring the outer loop
- Put the pieces together, preserving the nesting

18.1.2 Post-Test Loop

In some programming languages this is the **Do-While** loop. The code is run atleast once, and we only test after it has occured once.

This can be done in python by using the `break` command - This adds an additional exit point to the loop (other than the top condition of it). We add some test that can `break` from the loop at the bottom of the loop body, which means it can break if the condition is not met after a single loop, or consecutive loops.

Using too many breaks can cause problems - We need to be thoughtful when using it.

```
In [ ]: while True:
            number = float(input("Enter a positive number:"))
            if number >= 0:
                break  # Exit loop if number is valid
            else:
                print("That number is not positive")
```

19 Lecture 18

19.1 Simulation and design

One of the primary uses for computer programs is for simulation.

Pseudo Random Numbers: All random number generators work in much the same way - each generator has a **seed number** and this is used to generate some number.

19.2 Top-Down Design

This allows for the large problem to be made into smaller and simpler problems. Each smaller problem is then, itself, broken into even smaller problems until the little bits are all more manageable. The typical program uses the *input, process, output* pattern - This is then decomposed into smaller problems.

The name, parameters and expected return values of the fnctions have to be specified - This is called an API.

API: Functions with some kind of known input, process and output, which can be used without knowledge of the process' operations.

This all relates to the seperation of concerns - Having these *signatures* allows us to work on each piece of the program seperately.

Structure Chart: Think of the functions as boxes - What are the inputs and outputs? How do they interact with eachother. Whats calling what?

Abstraction: Take away the uneccessary bits of the design, ignore the details and let something else handle this.

Second level design Repeat the top level design for the next levels; This is to move from the main function, and then deal with the decomposition of the lower level functions until the entire program has been decomposed into their smallest units.

20 Lecture 19

20.1 Summary of the design process

1. Express the algorithm as a series of smaller problems
2. Develop an interface for each of the smaller problems
3. Detail the algorithm by expressing it in terms of its interfaces with the smaller problems
4. Repeat the process for each smaller problem

20.2 Bottom-Up implementation

Implementation is best done in small pieces - Start with the functions you know you need to put together.

20.2.1 Unit testing

Writing a function, and then testing it on its lonesome - Does it do what im expecting? For example: We can import our program and execute various reoutines/functions to ensure they work properly. To do this we need to ensure the reproudicability of the behaviour: The program must behave the same way each time it is executed.

After we have done unit testing, we must do **Integration Testing**. We need to ensure that other peoples codes in a mutliperson project works together.

21 Lecture 20

21.1 Summary of the design process

1. We start at the highest level of our structure and work our way down
2. At each level, we began with a general algorithm and refined it into precise code

This process is sometimes referred to as a step-wise refinement.

21.2 Prototyping and spiral development

Starting with a simplified view of what you want, then keep adding to it until the product is finished. This is called the **Agile** Methodology. This is a mini-cycle development methodology. Spiral development is useful when dealing with new or unfamiliar features or technology.

21.3 Search and recursion

Linear search: Goes through each of the elements of a list, will search until either the query item is found, or the end of the list is found.

Binary Search: Dividing the list by two and guessing either higher or lower, until the final result is found.

For a small number of list items, the linear search is better, however for a larger list: the binary search is the far superior option.

22 Lecture 21

Binary searches divide the problem in half, in order to make the problem much smaller - This is called **Divide and Conquer** .

Recurisve: Will call itself from inside itself, all the way down.

```
In [11]: def recfact(n):
             if n == 0:  # Terminating base case
                 return 1
             return n*(recfact(n-1))

         print(recfact(6))
```

```
720
```

The above example shows recursiveley defined factorial, but the same can be shown by using a loop in the following:

```
In [12]: def loopfact(n):
             output = 1

             for i in range(1, n+1):  # Base case is terminated by the end of a loop
                 output *= i
```

```
            return output

        print(loopfact(6))
```

```
720
```

Both of these methods give the same result, however the second option is more efficient: This is because the first option requires new function calls to be generated, which takes more CPU time than a cleverly thought out loop.

Base case: One side of the definition must be non recursive - When do we stop the recursion.

All good recursive definitions have:

- One or more base cases for which no recursion is applied
- All chains of recursion eventually end up at one of the base cases

23 Lecture 22

Iteration (looping) can sometimes be more difficult than recursion to express a problem. Many difficult problems using loops can be solved quite simply using recursion. Recursion can minimise the number of operations in some situations, such as exponentiation:

```
In [16]: def exponentiation(a, n):
             if n == 0:
                 return 1
             factor = exponentiation(a, n//2)

             if n%2 == 0:
                 return factor * factor
             return factor * factor * a

         print(exponentiation(2,5))
```

```
32
```

This takes logorithmic time to produce the answer. In this situation the iterative form will take linear time. With these two facts in mind, it only makes sense to sometimes use recursion over iteration. Recursion can often turn into a mess and often is best left to the iterative form.

23.1 Exam Discussion

- Total of 6 questions with a range of size and difficulty.
- Total of 120 marks: roughly a mark a minute
- Weighted 50% of the total marks.