Introduction to Scientific Computing
Topic 1: Basic 'Scalar' Types and Programming Techniques

*Lecturer: Dr Liew How Hui (*liewhh@utar.edu.my*)*
October 2023

1. Course Learning Outcomes According to UTAR Syllabus

CO1 perform vector and matrix operation using computer software..... Topic 1 and Topic 2 (Week 2)

CO2 plot graphs, curves, surfaces and contours using computer software ..... Topic 4 (Week 3--5)

CO3 write program scripts for mathematical software..... Topic 3 (Week 1)

CO4 apply computer software to solve system of linear equations, eigenvalue problems or matrix factorisation problems..... Topic 5 (Weeks 5--6)

CO5 apply computer software to perform curve fitting on a set of data..... Topic 6 (Week 7)

2. Course Arrangements and Assessments

* Week 1 and Week 2 (16 hours): Lectures and Practicals as usual.
** Practical assessment (3%) during practical class.

* Week 3:
** Monday --- Deepavali replacement holiday. No class
** Wednesday --- **Test** (14+3=17%. Q1: CO2, Topic 2; Q2: Topic 1, Topic 3)
** Thursday --- Dr Goh YK continues with Python visualisation.
** Thursday --- Announcement(?) of **Assignment** (17%)

* Week 4 and Week 5: Lecture and practical as usual by Dr Goh YK
** Dr Yong CK may start the class on Thursday

* Week 6:
** Practical Quiz 1 (8%) to be arranged by Dr Yong CK
** Submission of assignment report.
* Week 7:
** Monday --- Selangor Sultan Birthday Holiday. No class
** Practical Quiz 2 (8%) to be arranged by Dr Yong CK
* Total course hours = 48 hours

* In summary, Coursework (50%): Test (17%) + Assignment (17%) + Quizzes (16%)
* Final Exam (50%):
** Q1 (20%) and Q2 (20%) choose 1 only! --- CO2, Topic 1 and Topic 3
** Q3 (20%) --- CO1, Topic 2
** Q4 (20%) --- CO2, Topic 4
** Q5 (20%) --- CO4, Topic 5
** Q6 (20%) --- CO5, Topic 6

3. Introduction to ‘‘Scientific Computing’’

Scientific computing = numerical modelling and computation of (mostly) continuous models.

Continuous models:

* Calculus: for the formulation of ODEs, PDEs, integral equations, etc.;
* Ordinary Differential Equations (ODEs): from moving molecules to moving galaxies;
* Partial Differential Equations (PDEs): from the modelling of quantum particular, nanoparticles, climate modelling (for weather prediction) to evolution of cosmology.

Applications:

* data analysis and modelling
* machine learning: building ‘intelligent model’ from data for marketing, information management and business decision making
* Engineering modelling and calculations.

Objectives of scientific computing:

* Perform calculations correctly under proper discretisation and floating point arithmetic (i.e calculations with rounding in finite decimal places)
* Perform calculations efficiently (i.e. getting the result in less steps and also fast)
* Perform calculations with limited memory

Reference Books

* Jaan Kiusalaas, Numerical Methods in Engineering with Python 3, 3rd edition, Cambridge University Press 2013

* Hans Petter Langtangen, A Primer on Scientific Programming with Python, Springer 2016

4. Variables, Working Environments and Getting Help

Software for Scientific Computing: Anaconda Python

Variables are a combination starting with a capital or small letter follow by letters and/or numbers. It is used to refer to values/data/objects in computer.

```
Variable1 = 1 + 2 + 3.5
variable2 = "A string" + "another string"*2
var_3 = sin          # Small letter and
Var_3 = var_3(pi)    # capital letter are different!!!
```

General Commands for Working Environments under Command Prompt:

* Start a session: `python`
* End a session: `quit()`
* Load a module or package or function:
  `from math import sin, cos, pi`
* Reload a module: `importlib.reload(modulename)`
* Unload a package or module: `del modulename`
* **Working directory** in current session: The default directory for us to read and write a file.
  `import os; os.getcwd()`
* Get computer time:
  `import time; time.localtime(); time.asctime()`
* History for current session: `history()`
* List all variables in the current session: `dir()`
* Remove a variable: `del variablename`
* List the methods associated with an object: `dir(variable_name)`
* Check the type of a value / object: `type(value)`

Note that Python shell may not support session management in Windows. In Linux or MacOS, session management is automatic (i.e. the commands we keyed in are stored in .python_history, manual storing can be achieved by using the readline library:

```
readline.write_history_file('mypython.history')
```

In Jupyter, one can use `%hist -f my_history.py` to store the commands in a session.

Getting Help:

* **Online Help**: Google/Bing search for ‘‘python help’’ or ‘‘python documentation’’.
* Local Help and Documentation: Python is popular because it has very good libraries for text processing, scientific computing, etc. with help.

  `help(function_name)`

  can be used to find the description associated with function_name if it is defined. For example, help(sin) to get the short info about sine function.
* Numpy provides its specific help: `np.info(np.sin), np.lookfor('create array')`

5. Basic Data Types: Numbers and Arithmetic Operations, Containers.

String: `'Single-quoted'`, `"Double-quoted"`, `"""Multiline text"""` , etc.
Integer: `1234567890`, `0b101110`, `0o1235670`, `0x123456789ABCDEF0`, etc.
Boolean: `True`, `False`
Floating-point reals: `1., .1, 1.23, 123e-5, 5.45e+3`
List (square brackets): `[1,2,"a",True]`
Tuple (round brackets): `(1,2,"a",True)`

5.1. Characters, Strings, Texts

Usual characters: 0, 1, ..., 9, a, b, ..., z, A, ..., Z, punctuations, etc.

Control/Escape Characters: characters with backslash \, e.g. \n (new line), \t (tab), \%, etc.

String = an ordered collection of characters. It is normally expressed as quoted text.

Text = an ordered collection of strings usually separated by newlines. It is normally expressed as multiline text.

Operations related to strings:
* `"", ''` : empty string
* `len(s1)` : get the length of the string `s1`
* `s1 + s2` : joining strings
* `s1 * n` : repeat string `s1` `n` times
* `print('a string')` : Show the string to **terminal** (new line will automatically be added)
* `print('no newline', end='')`: turn off new line when printing
* `.lower()`, `.upper()`, `.capitalize()`: English/Latin related functions
* `int(s1)` : Convert a string in integer form to an integer
* `int(s2,base)` : Convert a string in base(<26) integer to an integer, e.g. `i2=int("123", 4)`
* `float("123.456e-1")` : convert a string to a floating point number

5.2. Booleans: True, False

Operations related to Booleans:
* `not b1` : negation
* `b1 and b2` : conjunction
* `b1 or b2` : disjunction
* `str(b1)` : convert the Boolean value to a string
* `print(b1)` : print the Boolean value

5.3. Integers: 0, 1, -1, 2, -2, 3, -3, ...

Python integers can be very very large depending on memory. Numpy integers or other programming languages (e.g. C, C++) integers are usually finite size. E.g. 16-bit (-2^{15} to $2^{15}-1$) 32-bit (-2^{31} to $2^{31}-1$) or 64-bit (-2^{63} to $2^{63}-1$).

Arithmetic Operations, Bit Operations, Relational Operations, string-related operations related to integers:
* `abs(x)` : absolute value of `x`
* `-4` : negation
* `3 + 4` : addition
* `3 - 4` : subtraction
* `3 * 4` : multiplication
* `4 // 3` : integer division; `4 % 3` : remainder
* `4 / 3` : floating-point number division
* `3 == 4, 3 != 4` : Check for (in)"equality"
* `3 < 4, 3 <= 4, 3 > 4, 3 >= 4`: Check for ordering
* `9 & 12` : Bit-and
* `9 | 12` : Bit-or

* `9 ^ 12` : Bit-xor
* `9 << 12` : Bit-shift-left
* `9 >> 12` : Bit-shift-right
* `str(i1)` : convert integer to decimal string
* `bin(i1)` : convert integer to binary string
* `oct(i1)` : convert integer to octal string
* `hex(i1)` : convert integer to hexadecimal string
* `"%6d" % i1` : **C-style formatting** for integer
* `"{:6d}".format(i1)`, `f"{i1:6d}"`: **C#-style formatting** for integer
* `bool(i1)`, `int(b1)`: Convert integer value to and from Boolean value
* `print(i1)` : print the integer value

5.4. Floating Point Numbers: 1.3, .1, 15., 1_2345.67, 1e-2, 3.5e2, -0.1e2, ...

Python (Real) Floating Point Number = IEEE 64-bit format binary representation to approximate a real number using 1 sign bit, 11-bit exponent and 52-bit mantissa.

Arithmetic Operations, Relational Operations, string-related operations related to floating-point numbers:
* Special constants : `from math import e, pi, tau, inf, nan`, `import math` (use `math.e`, etc.)
* Real function library : `from math import sin, cos, ...`
* abs(-1e2) : absolute value
* -4.0 : negation
* 0.1 + 0.2 : addition
* 0.1 - 0.2 : subtraction
* 0.1 * 0.2 : multiplication
* 0.1 / 0.2 : division
* 0.1 ** 0.2 : power $x^y = \exp(y \ln x)$.
* 0.1 + 0.2 == 0.3, 0.1 + 0.2 != 0.3: comparisons. Be careful with floating-point comparison, **rounding** can cause unexpected result!
* `0.1 < 0.2`, `0.1 <= 0.2`, `0.3 > 0.4`, `0.3 >= 0.4`: Check for ordering (be careful with rounding error issue)
* `str(f1)` : Convert a floating point value to a string
* `"%.6f" % f1` : **C-style formatting** for floating point value
* `"{:10.6f}".format(f1)`, `f"{f1:10.6f}"`: **C#-style formatting**

5.5. Complex Numbers: 1.3+0j, .1, 15.-2j, 45.67j-1_23, 1e-2j, -3.5e2j, ...

Python Complex Floating Point Number = A pair of real floating point numbers with similar properties to real floating point numbers.

* Representation: `1+2j`, `1.1e2-3.503j`, complex(3,5)
* Operation: abs(), -x, +, -, *, /, **, .conjugate(), .imag, `.real`, ==, !=
* Complex function library: `import cmath`

5.6. Formatting Numbers: Use C-style or C#-style formats

Example. Express the fine structure constant $\alpha = 7.297352568x10^{-3}$ in Python and print out its value in 6 decimal places and in scientific notation with 6 significant figures:

```
v=7.297352568e-3
print(f"{v:.6f}0v:.5e}")
```

5.7. List and Tuples

List and tuples are **collections / containers**, i.e. they are used to store zero or multiple basic data types and/or some complicated values/objects in Python.

Tuple

It is constructed using curved brackets (e.g. `(1,1.0,"Helo")`) or `tuple()` for programming construction.

There are virtually no methods associated with it.

* Use for value matching: `x1,x2,x3 = 3,4,5`
  Note: The round brackets can be ignored when we are assigning values.
* Swapping values: `x, y = y, x`

List

It is normally constructed using square bracket: `[1, 2, 3]` or `list()` for programming construction.

The important operations associated with it are `append`, `clear`, `copy`, `insert`, `sort`.

* For storing and the processing a list of values. E.g.

```
aString = input("Enter a list of integers (separated by a space): ")
anIntList = [int(i) for i in aString.split(" ")]
```

* A list is dynamic, so we can append or remove items from a list.
* We can assess list elements of `a` using `a[i]` where `i` is an integer in an appropriate range. Note that Python indexing starts from 0.
* A list of list of numbers can be used to represent matrix but it is slow for matrix arithmetic (so we usually work with Numpy arrays). E.g.

```
A = [[1,2],[3,4]]           # represents a 2x2 matrix A
B = [[8,6,7,9],[3,8,5,6]]   # represents a 2x4 matrix B
# The matrix multiplication A x B is
C = [
  [A[0][0]*B[0][0]+A[0][1]*B[1][0], A[0][0]*B[0][1]+A[0][1]*B[1][1],
   A[0][0]*B[0][2]+A[0][1]*B[1][2], A[0][0]*B[0][3]+A[0][1]*B[1][3]],
  [A[1][0]*B[0][0]+A[1][1]*B[1][0], A[1][0]*B[0][1]+A[1][1]*B[1][1],
   A[1][0]*B[0][2]+A[1][1]*B[1][2], A[1][0]*B[0][3]+A[1][1]*B[1][3]]
]
```

* We can use `sum(a)` on list of numbers `a` to calculate the sum of the elements in `a`. For example instead of writing

```
a = [1,3,5,7,9]
total = a[0] + a[1] + a[2] + a[3] + a[4]
```

  we can write

```
a = [1,3,5,7,9]
total = sum(a)
```

6. Script Files

A **script file** can be opened using **notepad** and we can read the content of the file to be a Python program.

Running a script file (file ends with `.py`)

A script file can directly run in Python shell. Under Linux's or MacOS's shell, or Windows' `cmd`, one needs to go to the working directory of the Python script and key in the following command followed by ''enter'' to *run* the script:

```
$ python my_python_script.py
$ python my_python_script.py > results.txt
```

The `results.txt` can be opened with word or insert into Word.

Jupyter notebook (file ends with `.ipynb`):

* It is a JSON file which need Jupyter notebook to read. Open using notepad shows us something different from a Python script.
* It needs to be opened by a Browser and commands will be send to Python shell through Browser.

Comments

A *comment* starts with #. It is a text line or paragraph inside a computer program with the intention of explaining a portion of the program.

For a long / multi-paragraph comments, we usually treat them as (doc-)strings which open and end with either `"""` or `'''`. We will put an `r` for 'raw strings' when we just want the \ to be a usual character rather than escape character.

Example. (Scripts for Solving Middle-School Maths --- Final Exam Oct 2018, Q2(a), CO3) The area of a triangle *ABC* can be calculated by the Heron's formula

$$|ABC| = \sqrt{s(s-a)(s-b)(s-c)}, \quad s = \frac{a+b+c}{2}$$

when the lengths of the three sides, $a = BC$, $b = AC$, $c = AB$ of the triangle *ABC* are given. The cosine rules of the triangle *ABC* are given below

$$a^2 = b^2 + c^2 - 2bc\cos A$$
$$b^2 = a^2 + c^2 - 2ac\cos B$$
$$c^2 = a^2 + b^2 - 2ab\cos C.$$

Given a triangle *PQR* with lengths $PQ = 4.5$cm, $PR = 3.5$cm and $QR = 7$cm. Write a **program script** to find and **print** the area of the triangle *PQR* and all the three angles *P*, *Q* and *R* in **degree**.

Sample Solution:

```
# Final Exam Oct 2018, Q2(a): A script to find triangle area and angles
a = 4.5
b = 3.5
c = 7.0
from math import sqrt, acos, degrees
s = (a+b+c)/2
Area = sqrt(s*(s-a)*(s-b)*(s-c))
print(f"The area of the triangle with a={a}, b={b}, c={c} is {Area}")
A = degrees(acos((b**2+c**2-a**2)/2/b/c))
B = degrees(acos((a**2+c**2-b**2)/2/a/c))
C = degrees(acos((a**2+b**2-c**2)/2/a/b))
print(f"Angle P = {C:8.4f} degree")
print(f"Angle Q = {B:8.4f} degree")
print(f"Angle R = {A:8.4f} degree")
```

Marks are awarded based on

* declaration of values for the three sides [1 mark]
* the appropriate import [1 mark]
* translation of mathematical formulae to computer instructions [6 marks]
* appropriate print commands [2 marks]

Example. (Theory of Interest SOA Exam FM Sample Questions Q1) Bruce deposits 100 into a bank account. His account is credited interest at an annual nominal rate of interest of 4% convertible semiannually.

At the same time, Peter deposits 100 into a separate account. Peter's account is credited interest at an annual force of interest of δ.

After 7.25 years, the value of each account is the same.

Calculate δ by writing a Python script.

Sample Solution:

```
IV_Bruce = 100
IV_Peter = 100
dur = 7.25  # duration in years
n   = 2     # semiannual
ir  = 0.04  # interest rate
FV_Bruce = IV_Bruce*(1+ir/n)**(n*dur)
# FV_Peter(delta) = IV_Peter*exp(dur*delta) = FV_Bruce
from math import log
delta = log(FV_Bruce/IV_Peter)/dur
print("delta = ", delta)
```

Try to type in the above code and get the answer for delta.

Exercise on AI: Try to ask the above two questions to **ChaptGPT** and **Wolfram Alpha** and see if they can generate Python scripts for you.

7. Functions

Functions are Python objects that usually takes in **zero or more parameters / values** and **do something** and may or may not **return values**.

A function can be abstractlly represented as

```
def f(x,y,z,...):
    statement_1
    statement_2
    ...
    return value
```

For simple functions, it is possible to use one of the following form:

```
def f(x): return an_expression_of x ...
f = lambda x: an_expression_of x ...
```

Reasons for defining functions:

(a) Not to repeat long programming statements!

(b) Breaking down a computation process with reasonable names.

On point (a), we will illustrate with the calculations of mean square error in machine learning.

```
a = [6,8,9];   b = [10,5,4]
c = [14,32,39]
d = [21,12,19]
e = [34,20,16]
f = [35,43,21]
mse1 = ((a[0]-b[0])**2 + (a[1]-b[1])**2 + (a[2]-b[2])**2) / 3
mse2 = ((c[0]-d[0])**2 + (c[1]-d[1])**2 + (c[2]-d[2])**2) / 3
mse3 = ((e[0]-f[0])**2 + (e[1]-f[1])**2 + (e[2]-f[2])**2) / 3
```

If we replace the 'repeating parts' of the above program using function, we have something shorter:

```
def mean_sq_err3(x, y):  # find mean square error of 3-element lists
    return ((x[0]-y[0])**2 + (x[1]-y[1])**2 + (x[2]-y[2])**2) / 3
mse1 = mean_sq_err3(a, b)
mse2 = mean_sq_err3(c, d)
mse3 = mean_sq_err3(e, f)
```

On point (b), we will use the formula of **sample standard deviation** for a list with 3 elements as an illustrative example:

$$sd3(x) = \sqrt{\frac{(x_1 - \overline{x})^2 + (x_2 - \overline{x})^2 + (x_3 - \overline{x})^2}{3}}$$

This means that the calculation popular standard deviation can be broken down to the functions 'square root', 'mean', 'sum', etc.

```
def mean3(x): return (x[0] + x[1] + x[2])/3.0
def sd3(x):   # popular standard deviation of 3-element list x
    xbar = mean3(x)
    total = sum([(x[0] - xbar)**2, (x[1] - xbar)**2, (x[2] - xbar)**2])
    from math import sqrt
    return sqrt(total/3)
```

Elementary built-in functions

Python only has very little built-in math functions for:

* numbers : `abs(x)`, `round(x)`, `pow(x,n)`
* many numbers : `min(`$x_1, x_2, \cdots, x_n$`)`, `max(`$x_1, x_2, \cdots, x_n$`)`
* list of numbers : `sum(x)`, `sorted(x)`
* list of booleans : `all([True,True,True])`, `any([False,True,False])`

We have to import math functions from the `math` module. Linear algebra and scientific functions are available from the Scipy's modules.

Example. Write down the Python command to calculate $\sin 10^o \sin 30^o \sin 50^o \sin 70^o$.

Solution: We must take note that numerical trigonometric functions only take radian, so we must convert degree to radian:

```
from math import sin, radians
sin(radians(10))*sin(radians(30))*sin(radians(50))*sin(radians(70))
```

Common Practices in Scientific Computing

It is a **common practise** to transform the problem we have to a **standard form** and then apply appropriate numerical methods to solve the problem.

* **Single-variable equation in standard form**:

$$F(x) = 0$$

* **Finding area under a function f:**

$$\int_a^b |f(x)|dx$$

* **Minimisation**:

$$\min_x F(x)$$

Example. (Single-variable equation) Solve the following nonlinear equation using Python

$$\sin(3x) = 2\cos(x).$$

Sample Solution: Note that the equation is not is standard form. We can subtract both sides with the expression on the right to obtain

$$\sin(3x) - 2\cos(x) = 0.$$

It is in standard form and we can solving using `scipy.optimize.fsolve`:

```
def F(x):
    from math import sin, cos
    return sin(3.0*x) - 2.0*cos(x)

import scipy.optimize
estimated_soln = scipy.optimize.fsolve(F,0.0)
print("The estimated solution is", estimated_soln)
```

Note that 0.0 is used as initial guess, other values may be OK or not OK. We have to try and see.

Example. (Solving Calculus Definite Integration) Create the Python commands to calculate

$$\int_0^{10} x^{x/2}dx + \int_0^{10} 0.9xdx.$$

It is not in standard form, we can choose to integrate the left expression by hand and the right expression using Calculus knowledge:

```
def f(x): return x**(x/2)

import scipy.integrate
estimate, error = scipy.integrate.quad(f, 0, 10)
answer = estimate + 0.9*(10**2/2.0)
```

Alternatively, we can write the integral into standard form since the lower and upper boundaries are the same.

$$\int_0^{10} \left\{ x^{x/2}dx + 0.9x \right\} dx.$$

We then write a Python script for the calculation:

```
def f(x): return x**(x/2) + 0.9*x

import scipy.integrate
answer = scipy.integrate.quad(f, 0, 10)
```

The answer is (61882.036798971014, 0.00028864395398611373).

7.1. Multiple Inputs and Outputs

Functions can take more than one input and may return multiple values.

```
# Multiple inputs: data1, data2
def method(data1, data2):
    ... perform calculations ...
    return (part1, part2)
```

Example. In maths, we can have functions with multiple variables. For example,

$$f(x, y) = \sin(xy)$$

It be expressed in Python as a function with multiple inputs:

```
from math import sin
def f(x,y): return sin(x*y)
```

In maths, we can also have functions with multiput outputs. For example, a function which represents an ellipse:

$$g(t) = (3\cos(t), 2\sin(t))$$

It be expressed in Python as a function with multiple outputs:

```
from math import sin, cos
def g(t): return (3*cos(t), 2*sin(t))
```

7.2. Variable Arguments

When a function or a method of the same name has different number of inputs, it is said to have variable arguments. For example, the summation can have 'any number of inputs':

```
def total(a,b): return a+b
def total(a,b,c): return a+b+c
```

No one likes to write functions like this by repeat copy and paste but in some programming languages, this is what they implement. Fortunately, Python provides the **variable length argument lists** and together with the for loop which we will learn allows us to implement a function with any number of inputs nicely:

```
def total(first, *rest):
    t = first
    for item in rest: t += item
    return t
```

The **variable length argument lists** is used in the Python `print` function as follows:

```
print("Adding", 1, "and", 2, "gives us", 1+2)
```

However, there are functions with 'sort of fixed' number of inputs but less values can be passed to the function by assuming **default values**:

```
def afunction(par1, par2=some_default_value):
    ...
```

There is also a syntax to use the **keyword-ed argument lists** (which uses the Dictionary data structure):

```
def bfunction(farg, **kwargs):
    ...
    for key in kwargs:
        do something to the value kwargs[key]
        ...
    ...
```

The `afunction` and `bfunction` are used in more advanced programming context.

7.3. Positional-only parameters (Specific to Python)

According to https://www.python.org/dev/peps/pep-0570/, a function definition may look like:

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
      -----------    ----------     ----------
        |             |                   |
        |        Positional or keyword    |
        |                                 - Keyword only
         -- Positional only
```

where `/` and `*` are optional. If used, these symbols indicate the kind of parameter by how the arguments may be passed to the function: positional-only, positional-or-keyword, and keyword-only. Keyword parameters are also referred to as named parameters.

Note that **this feature is only available in Python 3.8 and above**.

```
def f(a,b,/,c,d,*,e,f): print(a,b,c,d,e,f)
f(10,20,30,40,50,f=60)
f(10,20,30,d=40,e=50,f=60)
```

8. Input-Output and File Types

An **input** refers to anything that computer ‘‘gets’’ data and store into computer memory. So ‘‘keyboard’’ is an input, ‘‘mouse’’ is an input, ‘‘tablet’’ is an input, a ‘‘computer file’’ on our desktop is an input, etc.

An **output** refers to anything that computer ‘‘displays’’ or ‘‘stores’’ data from computer memory. So a ‘‘computer monitor’’ is an output, a ‘‘printer’’ is an output, a ‘‘computer file’’ on our desktop is an output, etc.

To illustrate a simple input-output between ‘‘keyboard’’ and ‘‘screen’’ (is ‘‘teletype terminal’’ a more precise term?), we will play with the following Python commands:

```
width = 70
pre = "0*2 + "*"*width + "0" + " "*(width-2) + "*0"
post = "*0" + " "*(width-2) + "*0 + "*"*width
greet = "0nter your name: "
name = input(greet); print(pre+("Hello "+name).center(width-2)+post)
```

The command `input` in line 5 reads from ‘‘keyboard’’, displays what you type on ‘‘screen’’ and stores the string into the variable `name`.

However, there is a major problem with ‘‘keyboard’’ and ‘‘screen’’ --- the data which is keyed in and displayed on ‘‘screen’’ will disappeared once we turn off the computer. A ‘‘computer file’’ is something that will remain in computer even after we have turned off a computer and is hence the best choice for storing and retrieving data related to scientific computing.

Similar to humans speaking many different languages, computer files also ‘‘storing’’ many different file formats. All file formats can be categorised into two file types --- *text* file type and *binary* file type. The basic operations associated with a file are ‘‘open’’, ‘‘read’’, ‘‘write’’ and ‘‘close’’. The ‘‘save’’ function is the same as opening file for writing and after writing data into it, close it.

Open text file for reading:	`fp=open("f", "rt")`
Read text file:	`x=fp.readlines()`
Open text file for writing:	`fp=open("g", "wt")`
Write text file:	`fp.writelines(x)`
Open binary file for reading:	`fp=open("f", "rb")`
Read binary file:	`M=fp.read()`
Open binary file for writing:	`fp=open("f", "wb")`
Write binary file:	`fp.write(x)`
Close file:	`fp.close()`

9. If-else-then or Selective Statements

The ability for a computer program to ‘‘select’’ or ‘‘choose’’ is crucial because it allows us to implement ‘‘decision’’. In Python, ‘‘selection’’ is achieved by the various forms of the ‘‘if’’ selective statements. A list of selective statements are given below.

If only:

```
if condition:
    do_something_block
```

If-else statement:

```
if condition:
    do_something_block
else:
    do_otherthing_block
```

One-line form (ternary operator):

```
res = true_result if condition else false_result
```

Multiple conditions:

```
if condition1:
    do_task1_block
elif condition2:
    do_task2_block
elif condition3:
    do_task3_block
else:
    do_default_task_block
```

Example. (If-else statement and One-line form) Consider a very simple function:

$$H(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases}$$

Multiline form:

```
def H(x):
    if x < 0.0:
         return 0.0
    else:
         return 1.0
```

One-line form:

```
def H(x): return 0.0 if x < 0.0 else 1.0
```

Example. (Final Exam Sept 2012, Q3(a) --- Multiple-conditions) Consider the quadratic equation $ax^2 + bx + c = 0$. The solution to this equation is

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

The term ‘‘$b^2 - 4ac$’’ is called the discriminate of the equation. The nature of the discriminate determines the number and type of the roots as follows:

discriminate value	number and type of roots
positive	2 distinct real roots
negative	2 complex roots
zero	1 repeated root

Write a Python program to solve for the roots of a quadratic equation. The program should:

* read the input values of *a*, *b*, and *c*;
* calculate the roots; and
* display the outputs, including a statement about the type of the roots, i.e. ‘‘There are 2 distinct real roots.’’

The output should be displayed as below.

```
This program solves for the roots of a quadratic equation of
  the form ax^2 + bx + c = 0.
input the value of the coefficient "a": 1
input the value of the coefficient "b": 2
input the value of the coefficient "c": 3
the discriminate is -8.000000
the equation has two complex roots
x1 = -1.000000 + 1.414214 i
x2 = -1.000000 - 1.414214 i
```

Sample Solution using Multi-condition statement:

```
from math import sqrt
print("""This program solves for the roots of a quadratic equation of
  the form ax^2 + bx + c = 0.""")
a = float(input('input the value of the coefficient "a": '))
b = float(input('input the value of the coefficient "b": '))
c = float(input('input the value of the coefficient "c": '))
discriminate = b**2 - 4*a*c
print("the discriminate is %.6f" % (discriminate))
if discriminate > 0:
    print("the equation has two distinct real roots")
    sq = sqrt(discriminate)
    print("x1 = {x:.6f}".format(x=(-b+sq)/2.0/a))
    print("x2 = {x:.6f}".format(x=(-b-sq)/2.0/a))
elif discriminate < 0:
    print("the equation has two complex roots")
    impart = sqrt(-discriminate)/2.0/a
    repart = -b/2.0/a
    print("x1 = {re:.6f} + {im:.6f} i".format(re=repart, im=impart))
    print("x2 = {re:.6f} - {im:.6f} i".format(re=repart, im=impart))
else:
    print("the equation has one repeated root")
    print("x1 = x2 = {x:.6f}".format(x=-b/2.0/a))
```

Example. Given a cubic equation

$$a_0x^3 + a_1x^2 + a_2x + a_3 = 0 \quad => \quad x^3 + ax^2 + bx + c = 0.$$

Let $x = t - \frac{b}{3a}$ and

$$\begin{cases} t = x + \dfrac{b}{3a}, \\ p = \dfrac{3ac - b^2}{3a^2}, \\ q = \dfrac{2b^3 - 9abc + 27a^2d}{27a^3}, \\ h = \dfrac{p^3}{27} + \dfrac{q^2}{4}. \end{cases}$$

* If $h = 0$, the cubic has multiple roots

* $p = 0$ (together with $h = 0$) $=> q = 0 => x_1 = x_2 = x_3 = -\frac{b}{3a}$

* $p \neq 0 => \ t_1 = \frac{3q}{p}, t_2 = t_3 = -\frac{3q}{2p} => x_k = t_k - \frac{b}{3a}, k = 1, 2, 3.$

* If $h > 0$ and p, q are real, the cubic has two complex roots and one real root based on the Cardano's formulae:

$$t_1 = \sqrt[3]{-\frac{q}{2} + \sqrt{\frac{q^2}{4} + \frac{p^3}{27}}} + \sqrt[3]{-\frac{q}{2} - \sqrt{\frac{q^2}{4} + \frac{p^3}{27}}}$$

$$t_2 = \frac{-1 + \sqrt{3}i}{2}\sqrt[3]{-\frac{q}{2} + \sqrt{\frac{q^2}{4} + \frac{p^3}{27}}} + \frac{-1 - \sqrt{3}i}{2}\sqrt[3]{-\frac{q}{2} - \sqrt{\frac{q^2}{4} + \frac{p^3}{27}}}$$

$$t_3 = \frac{-1 - \sqrt{3}i}{2}\sqrt[3]{-\frac{q}{2} + \sqrt{\frac{q^2}{4} + \frac{p^3}{27}}} + \frac{-1 + \sqrt{3}i}{2}\sqrt[3]{-\frac{q}{2} - \sqrt{\frac{q^2}{4} + \frac{p^3}{27}}}.$$

* If $h < 0$, the cubic has 3 distinct roots based on the trigonometric formulae:

$$t_k = 2\sqrt{-\frac{p}{3}}\cos\left[\frac{1}{3}\arccos\left(\frac{3q}{2p}\sqrt{\frac{-3}{p}}\right) - \frac{2\pi(k-1)}{3}\right] \quad \text{for } k = 1, 2, 3.$$

Sample Implementation of Cubic Equation Solver:

```
def cubic(poly):
    if len(poly) == 4:
        a, b, c, d = poly
    else:
        print("Coefficients", poly, "are not cubic")
        return None

    if a == 0: return quadratic([b,c,d])

    # https://en.wikipedia.org/wiki/Cubic_equation
    p = ((3*c / a) - (b**2 / a**2)) / 3
    q = ((2*b**3 / a**3) - (9*b*c / a**2) + (27*d / a)) / 27
    # h is the discriminant of the depressed cubic t^3 + tp + q = 0
    # where t = x + b/(3a)
    h = (q**2 / 4) + (p**3 / 27)

    #
    # Assuming all the coefficents are real
    # h < 0: the cubic has 3 real roots
    # h = 0: the cubic has multiple root
    # h > 0: the cubic has 1 real root and 2 complex conjugate roots
    #
    from math import sqrt, sin, cos, acos, pi
    b_3a = b / (3*a)

    if p == q == h == 0: # all 3 roots are real and equal
        return (-b_3a,)*3

    # Cardano's formula (https://en.wikipedia.org/wiki/Cubic_equation)
    if h > 0: # only 1 root is real
        e = -(q/2) + sqrt(h)
        s = -((-e)**(1/3)) if e < 0 else e**(1/3)
        e = -(q/2) - sqrt(h)
        u = -((-e)**(1/3)) if e < 0 else e**(1/3)

        t1 = s + u
        t2 = complex(-1, sqrt(3))/2*s + complex(-1,-sqrt(3))/2*u
        t3 = complex(-1,-sqrt(3))/2*s + complex(-1, sqrt(3))/2*u
```

```
# Trigonometric formula (https://en.wikipedia.org/wiki/Cubic_equation)
if h <= 0: # all 3 roots are real
    e  = sqrt(-p/3)
    t1 = 2*e*cos(acos((3*q)/(2*p)/e)/3)
    t2 = 2*e*cos(acos((3*q)/(2*p)/e)/3 - 2*pi/3)
    t3 = 2*e*cos(acos((3*q)/(2*p)/e)/3 - 4*pi/3)

x1, x2, x3 = t1 - b_3a, t2 - b_3a, t3 - b_3a
x1 = complex(0,x1.imag) if abs(x1.real) < 1e-16 else x1
x2 = complex(0,x2.imag) if abs(x2.real) < 1e-16 else x2
x3 = complex(0,x3.imag) if abs(x3.real) < 1e-16 else x3
x1 = x1 if abs(x1.imag) > 1e-16 else x1.real
x2 = x2 if abs(x2.imag) > 1e-16 else x2.real
x3 = x3 if abs(x3.imag) > 1e-16 else x3.real

return x1, x2, x3
```

10. Loops

Loops are applied to solve problems with repetition, for example, if there are many mathematical problems involving the calling of a function multiple times:

```
f(1) + f(2) + f(3) + ... + f(1000)
```

It is unwise to copy and paste 1000 times!

Python provides several kinds of ‘‘loops’’ --- for loop, while loop and recursion to solve ‘looping’ problem.

10.1. For Loop

```
for variable in somerange:
    block
```

The `somerange` is usually (suppose `n` is nonnegative)

```
range(start,end,step): start, start+step, ..., before ‘end-step’
range(20,1,-3): 20, 17, 14, 11, 8, 5, 2
range(1,10): 1, 2, 3, 4, 5, 6, 7, 8, 9
range(n)   : 0,1,2,...,n-1  (Python starts from 0 by default)
```

Example. (Final Exam Sept 2016, Q3(a) with modification) Consider the alternating series

$$S = \sum_{n=1}^{k} (-1)^{n+1} \frac{1}{n\ln(n+1)}.$$

Write a Python expression to calculate S for $k = 100$.

Sample Solution: In this example, we can see that we have

$$f(n) = \frac{(-1)^{n+1}}{n\ln(n+1)}$$

and we need to calculate

```
f(1) + f(2) + f(3) + ... + f(100)
```

Two possible Python implementations are illustrated:

```
from math import log
def f(n): return (-1)**(n+1)/n/log(n+1)

def S_1(k):         # This is similar to C, C++, C#, etc.
    total = 0.0
    for n in range(1,k+1):
        total += f(n)

def S_2(k):         # This is specific to Python
    return sum(f(n) for n in range(1,k+1))

print("S(100) =", S_1(100))
```

Example. (Final Exam Sept 2014, Q2(c) with modification) Write a Python code to calculate the following summation

$$3(2 + 1) + 4(3 + 2 + 1) + 5(4 + 3 + 2 + 1) + 6(5 + \ldots + 1) + \cdots + 1000(999 + \ldots + 1).$$

Sample Solution: If we try to match the pattern, we espect

$$f(n) = n((n-1) + (n-2) + \cdots + 1)$$

The summation in the question can be expressed as

$$f(3) + f(4) + \cdots + f(1000)$$

```
def f(n):
    return n*sum(range(1,n))  # n*(1 + ... + n-1)

total = sum(f(i) for i in range(3,1001))
```

Example. (Shannon Entropy) Suppose we want to transmit English text and assume the 26 characters plus a space has equal probability, the **entropy** under binary transmission system is

$$H(\textit{single character}) = \log_2(27) = \log_2(\frac{1}{1/27}) = -\log_2(\frac{1}{27})$$

Here the 1/27 is the probability of one character. When we are transmitting multiple characters or words of different probabilities p_i, the entropy is the **average of entropy**:

$$H(S) = -\sum_i (p_i \times \log_2 p_i).$$

The implementation in Python is

```
def H(S):
    from math import log2
    H1 = lambda p: 0.0 if x==0 else -p*log2(p)
    return sum(H1(p) for p in S)
```

For loop is used in programming for many things apart from the mathematical summation demonstrated earlier.

Example. (Final Exam Sept 2015, Q4(b) --- Weighted Mean in Statistics) A weighted mean is used when there are varying weights for the data values. For a data set given by $x = x_1, x_2, x_3, \ldots, x_n$ and corresponding weights for each x_i, $w = w_1, w_2, w_3, \ldots, w_n$, the weighted mean is

$$\frac{\sum_{i=1}^{n} x_i w_i}{\sum_{i=1}^{n} w_i}.$$

Write a function that will receive two vectors as input arguments: one for the data values and one for the weights and will return the weighted mean.

Sample Solution using For Loop:

```
def weigthed_mean(data, weights):
    numer = 0.0
    denom = 0.0
    for i in range(len(weights)):
        numer = numer + weights[i]*data[i]
    for i in range(len(weights)):
        denom = denom + weights[i]
    return numer/denom
```

Example. (Final Exam Oct 2019 Q1(b) --- Speed up computation) https://en.wikipedia.org/wiki/Horner's_method is a polynomial evaluation method expressed by

$$p(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + \cdots + x(a_{n-1} + xa_n)))).$$

Implement the Horner's method as a Python function `horner(coeffs,x)` which takes in the coefficients $a_0, a_1, \ldots, a_n$ of the polynomial $p(x)$ as `coeffs` and a value `x` and then returns the value of the polynomial $p(x)$ at `x`.

Sample Solution:

```
# https://rosettacode.org/wiki/Horner's_rule_for_polynomial_evaluation
def horner(x,coeffs):
    deg = len(coeffs)
    y = coeffs[deg-1]
    for j in range(N-2,-1,-1):
        y = coeffs[j] + x*y
    return y
```

Example. (Estimating the errors for equations) Let us investigate the nonlinear equation mentioned earlier

$$\sin(3x) = 2\cos(x)$$

using for loop over the range 0<=x<=π.

Sample Solution:

```
from math import sin, cos, pi
def f(x): return sin(3*x) - 2*cos(x)

N = 10  # trying 10 points over [0,π]
for i in range(N):
    x = i*pi/N
    print("x={x:.4f}, y={y: 12.8f}".format(x=x, y = f(x)))
```

Example. (Discrete Mathematics, Tutorial 1, Q1(a)(iv)) Generate the truth table of the statement $\sim s \rightarrow (p \wedge (q \vee r))$ where *p, q, r, s* are atomic statements.

Sample Solution:

```
def v(val): return "T" if val else "F"
def statement1(p,q,r,s):
    return s or (p and (q or r))
print("p | q | r | s | statement")
print("--+---+---+---+-----------")
for p in [True, False]:
    for q in [True, False]:
        for r in [True, False]:
            for s in [True, False]:
                val = statement1(p,q,r,s)
                print(f"{v(p)} | {v(q)} | {v(r)} | {v(s)} | {v(val)}")
```

Example. (Final Exam Oct 2022 Q1(b)) Write a Python script without importing any modules but only using appropriate data representation and for loops to generate a distance matrix for the 3-D points P_i, $i = 1, 2, 3, 4, 5$ in Table 1.1.

Point	x	y	z
P_1	2	3	5
P_2	2	5	3
P_3	6	4	2
P_4	5	3	4
P_5	3	5	4

Table 1.1: 3-D points.

Define a function to calculate the Euclidean distance between two 3-D points in your Python script and then go through all the pairs of the points P^1 to P_4 to generate a distance table in Table 1.2.

```
0.000000   2.828427   5.099020   3.162278   2.449490
2.828427   0.000000   4.242641   3.741657   1.414214
5.099020   4.242641   0.000000   2.449490   3.741657
3.162278   3.741657   2.449490   0.000000   2.828427
2.449490   1.414214   3.741657   2.828427   0.000000
```

Table 1.2: Distance matrix of the 3-D points.

Sample Solution using Double For-loop:

```
points = [(2,3,5), (2,5,3), (6,4,2), (5,3,4), (3,5,4)]
print(points)                                        # [2 marks]

def euclid_dist(pt1, pt2):                           # [1 mark]
    return ((pt2[0]-pt1[0])**2 + (pt2[1]-pt1[1])**2
              + (pt2[2]-pt1[2])**2)**0.5             # [2 marks]

for pt1 in points:                                   # [1 mark]
    for pt2 in points:                               # [1 mark]
        dist = euclid_dist(pt1, pt2)
        print(f"{dist:10.6f} ", end="")  # same row
    print()  # Print a new line                       # [1 mark]
```

Example. (Final Exam Oct 2019 Q1(c)) In the late 1960's, book publishers realised that they needed a uniform way to identify all the different books that were being published throughout the world. In 1970 they came up with the International Standard Book Number system. Every book, including new editions of older books, was to be given a special number, called an ISBN, which is not given to any other book. The check digit x_{10} of a 10-digit ISBN $x_1x_2 \cdots x_9x_{10}$ is given by the formula:

$$x_{10} = (11 - (10x_1 + 9x_2 + 8x_3 + 7x_4 + 6x_5 + 5x_6 + 4x_7 + 3x_8 + 2x_9 \ mod\ 11)) \ mod\ 11.$$

If $x_{10} = 10$, it is represented as 'X'. Write a Python **program script** to implement the function `check-digit10(isbn)` which takes an ISBN of the form $x_{1x_2} \cdots x_9$ as a string of length 9 and returns a digit x_{10} as a string of length 1.

* Correct definition of function [1 mark]
* Proper use of for loop [1 mark]
* Demonstrate correct translation of mathematical formula to computer program [3 marks]

Sample Solution:

```
def check_digit_10(isbn):
    isbn = list(isbn.replace('-','').replace('?',''))
    assert len(isbn) == 9
    x10 = 0
    for i in range(len(isbn)):
        x10 += (10-i)*int(isbn[i])
    x10 = (11-x10) % 11
    return str(x10) if x10 != 10 else 'X'

print(check_digit_10('0-8044-2957'))
print(check_digit_10('0-85131-041'))
```

Example. In the late 1960's, book publishers realised that they needed a uniform way to identify all the different books that were being published throughout the world. In 1970 they came up with the *International Standard Book Number* system. Every book, including new editions of older books, was to be given a special number, called an ISBN, which is not given to any other book. ISBN is changed from a 10-digit system to 13-digit system since 1 January 2007. The check digit of the ISBN-13 is given by the formula (see https://en.wikipedia.org/wiki/International_Standard_Book_Number)

$$x_{13} = \begin{cases} r, & r < 10, \\ 0, & r = 10 \end{cases}$$

where

$$r = (10 - (x_1 + 3x_2 + x_3 + 3x_4 + x_5 + 3x_6 + x_7 + 3x_8 + x_9 + 3x_{10} + x_{11} + 3x_{12}) \ mod\ 10).$$

Provide two implementations of the ISBN-13 check digit as a Python function `check(isbn)` which takes in an array of digits and return a check digit, one implementation which uses for loop and one implementation which uses Numpy array. Find the check digit for `978-0-306-40615-?`.

Sample Solution using For Loop:

```
    # http://code.activestate.com/recipes/498104-isbn-13-converter/
    def check_digit_13(isbn):
        isbn = list(isbn.replace('-','').replace('?',''))
        assert len(isbn) == 12
        sum = 0
        for i in range(len(isbn)):
            c = int(isbn[i])
            w = 3 if i % 2 else 1
            sum += w * c
        r = 10 - (sum % 10)
        if r == 10: return '0'
        else: return str(r)
```

Example. (Magic Square in Combinatorics) A **magic square** is an arrangement of distinct numbers (i.e., each number is used once), usually integers, in a square grid, where the numbers in each row, and in each column, and the numbers in the main and secondary diagonals, all add up to the same number, called the "magic constant". A magic square has the same number of rows as it has columns, and in conventional math notation, n stands for the number of rows (and columns) it has and is called the 'order of the magic square'. Thus, a magic square of order *n* contains n^2 numbers.

A magic square can be denoted as a list of *n* list with *n* numbers. Write a Python function to **check if an object is a magic square** and then find an algorithm to generate magic squares from the Internet. Can we ask AI to write such a script for us?

Sample Solution: The following Python function can be used to check a magic square encoded in the form of a list of lists (rows).

```
    # https://www.w3resource.com/python-exercises/math/python-math-exercise-20.php
    def is_magic_square(obj):
        n = len(obj)
        sums_from_every_row = [sum(row) for row in obj]
        sums_from_every_col = []
        for j in range(n):
            col = [obj[irow][j] for irow in range(n)]
            sums_from_every_col.append(sum(col))
        #Two diagonals
        diag1 = diag2 = 0
        for i in range(n):
            diag1 += obj[i][i]
            diag2 += obj[i][n-i-1]
        return diag1==diag2 and \
            all([sums_from_every_row[i]==diag1 for i in range(n)]) and \
            all([sums_from_every_col[i]==diag1 for i in range(n)])

    m=[[7, 12, 1, 14], [2, 13, 8, 11], [16, 3, 10, 5], [9, 6, 15, 4]]
    print(is_magic_square(m))
    print(is_magic_square([[2, 7, 6], [9, 5, 1], [4, 3, 8]]))
    print(is_magic_square([[2, 7, 6], [9, 5, 1], [4, 3, 7]]))
```

The following **magic square generation algorithm** is popular in the Internet.

```
# https://www.codesansar.com/python-programming-examples/generate-magic-square.htm
# https://scipython.com/book/chapter-6-numpy/examples/creating-a-magic-square/
def magic_sqr_method1(n):
    if n % 2 == 0: return []  # Only works with odd n
    magic_square = []
    for i in range(n):
        row = [0]*n
        magic_square.append(row)
    cnt, i, j = 1, 0, n//2
    while cnt <= n**2:
        magic_square[i][j] = cnt
        cnt += 1
        newi, newj = (i-1)%n, (j+1)%n
        if magic_square[newi][newj]:
```

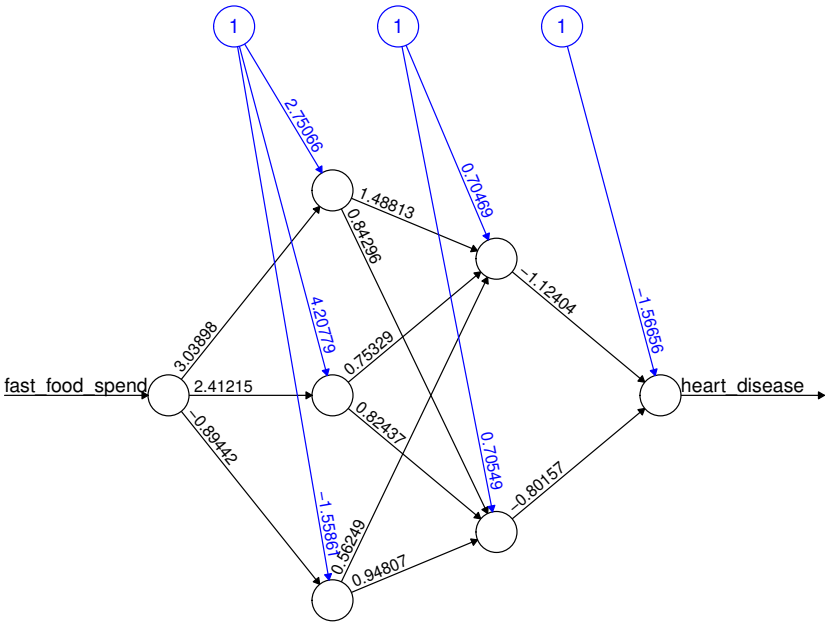
```
            i += 1
        else:
            i, j = newi, newj
    return magic_square
```

Example. Consider the heart disease data from https://bookdown.org/brianmachut/uofm_analytics_r_hw_sol_2/logreg.html by analysing the relation between X=fast_food_spend and Y=heart_disease.

One mathematical model for fitting the data is called logistic regression model:

$$P(Y = 1|X = x) = \frac{1}{1 + \exp(-(-10.651330614 + 0.002199567x))}$$

Another mathematical model for fitting the data is the 3-layer neural network is shown below:



Error: 160.952352 Steps: 25

$$\begin{cases} u_1 = \dfrac{1}{1 + \exp(-(2.75065899 + 3.03897930x))} \\ u_2 = \dfrac{1}{1 + \exp(-(4.20779155 + 2.41214781x))} \\ u_3 = \dfrac{1}{1 + \exp(-(-1.55861321 - 0.89442128x))} \end{cases}$$

$$\begin{cases} v_1 = \dfrac{1}{1 + \exp(-(0.70469462 + 1.48813242u_1 + 0.75329235u_2 + 0.56249011u_3))} \\ v_2 = \dfrac{1}{1 + \exp(-(0.70548672 + 0.84296074u_1 + 0.82436993u_2 + 0.94807080u_3))} \end{cases}$$

$$y = \frac{1}{1 + \exp(-(-1.56656128 - 1.12404432v_1 - 0.80157036v_2))}$$

By using the knowledge you have learned so far, write a Python script to read heart_data.csv and express the logistic regression model as Python function. Is it easy to implement the neural network model in Python?

Sample Python Script Solution:

```
fp = open("heart_data.csv")
colnames = fp.readline().strip().split(",")
data = fp.readlines()
p = len(colnames)
# We analyse using column 1 (heart_disease, output) and column 3 (fast_food_spend, inp
col1 = []
col3 = []
for row in data:
    items = row.strip().split(",")
    col1.append(float(items[0]))
    col3.append(float(items[2]))
from matplotlib.pylab import plt
plt.plot(col3,col1,"*")
plt.show()

# Expressing logistic regression model as Python functions
def log_reg(x):
    from math import exp
    return 1.0/(1.0+exp(-(-10.651330614 + 0.002199567*x)))
```

Interrupting a Loop: break, continue

The `break` statement is used to stop a loop early.

Application of break statement: If the error of a calculation is accurate enough, break out from the loop to improve accuracy. E.g. bisection method, newton method, ...

The `continue` statement is used to skip some statements in a loop and continue with the next counter.

10.2. While Loop

In contrast to ‘‘for loop’’, we seldom use ‘‘while loop’’. It is used when we don’t know when to stop. A ‘‘while loop’’ statement may often be augmented with `break` statement to jump out of the loop.

```
while condition1:
    block
    if condition2: break
    other block  (won’t be executed if condition2 is true)
```

Example. (Final Exam Sept 2012, Q1(b) with modification) Write a Python script to generate a sequence of random number and determine the number of attempt it takes to obtain the first random number that is in between 0.5 and 0.55.

Sample Solution using While Loop:

```
import numpy as np
count = 0
while True:
    anum = np.random.rand()
    count = count + 1
    if 0.5 < anum < 0.55:
        print("Number of attempt taken =", count)
        break  # exit while loop
```

Example. (Final Exam Sept 2013, Q5(b) with modification) Write a Python script to determine the greatest value of *n* that can be used in the sum

$$2 + 4 + 6 + 8 + \cdots + 2n$$

and get a value of less than 200.

Solution: If we use Python’s `range` and `sum` function, we can solve it easily with a `while` loop as follows:

```
thesum = n = 0
while thesum < 200:
    n = n+1
    thesum = sum(range(2,2*n+1,2))
    greatest_n = n-1
print('The greatest value of n such that 2+4+...+2n<200 is', greatest_n)
```

However, this is very inefficient because in line 4, we are summing numbers again and again. A more efficient Python program is as follows:

```
thesum = n = 0
while thesum < 200:
    n = n+1
    thesum += 2*n
    greatest_n = n-1
print('The greatest value of n such that 2+4+...+2n<200 is', greatest_n)
```

The greatest value of n such that $2 + 4 + \ldots + 2n < 200$ is 13.

Example. (Final Exam Sept 2017, Q3(c) with modification) Write a Python script to calculate the following summation not exceeding 100. Script must give total sum and the last element as output.

$$\frac{4}{5} + \frac{5}{6} + \frac{6}{7} + \frac{7}{8} + \frac{8}{9} + \ldots.$$

Sample Solution:

```
n = 0
thesum = nextterm = 0.0
while thesum < 100.0:
    nextterm = (n+4.0)/(n+5.0)
    thesum += nextterm
    n = n+1
print('Total sum is', thesum-nextterm)
print("and the last element at output is", n+3, "/", n+4)
```

Total sum is 99.82862321536608 and the last element at output is 107/108.

10.3. Recursion

Recursion is a theoretical way to do **looping**.

Recursion stands for ''the act of a function calling itself to do things''. This is a technique for estimating the time or space complexity of computer algorithm related to trees and graphs.

Example. The mathematical definition of Fibonacci numbers are given by

$$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}, \quad n = 2, 3, \ldots$$

The recursive function implemetation of Fibonacci is shown below.

```
def fib(N):
    """
    return the N-th Fibonacci number.
    """
    if N<=0:
        return 0
    elif N==1:
        return 1
    else :
        return fib(N-1) + fib(N-2) # can use up a lot memory
```

Example. Consider a variation of the recurrence relation found in 2017 West Australia Applications Exam:

$$T_{n+1} = 4T_n - 3, \quad T_1 = 2.$$

Implement this recurrence relation as a Python function `T(n)`. In your implementation, return `None` if `n` is less than 1. **Demonstrate** the workings of the function by writing a Python script to display `T(1)`, `T(2)`, ..., `T(10)`. Write down the value of `T(4)`.

Sample Solution:

```
def T(n):
    if n < 1:
        return None
    elif n == 1:
        return 2
    else:
        return 4*T(n-1)-3

for n in range(1,11):
    print(f"T({n}) = {T(n)}")
```

Example. (Final Exam Sept 2015, Q2(d)) The Taylor series of a cosine function at $x = 0$ is

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots.$$

Write a recursive function that computes and approximation of cosine.

Sample Solution using Recursion:

```
def cosine_approx(n,x):
    if n<=0:
        return 1.0
    elif n%2==1:  # n is odd
        return cosine_approx(n-1,x)
    else:         # n is even
        highest_order_term = 1.0
        for i in range(2,n+1):
            highest_order_term *= x/i
        if (n/2)%2 == 1: highest_order_term *= -1
        return highest_order_term + cosine_approx(n-2,x)
```

Recursion is a method to perform calculation for programming languages which does not have for loop or while loop.