UECM1703 Introduction to Scientific Computing
Topic 1: Introduction, Scripting and Programming

Lecturer: Dr Liew How Hui (`liewhh@utar.edu.my`)

Oct 2022

Course Learning Outcomes According to UTAR Syllabus:

1. perform vector and matrix operation using computer software; .... Topic 1 and Topic 2 (Week 2)
2. plot graphs, curves, surfaces and contours using computer software; .... Topic 4 (Week 3–5)
3. write program scripts for mathematical software; .... Topic 3 (Week 1)
4. apply computer software to solve system of linear equations, eigenvalue problems or matrix factorisation problems; .... Topics 5 (Week 5–6)
5. apply computer software to perform curve fitting on a set of data. .... Topic 6 (Week 7)

Contents

§1.1 Course Arrangements and Assessments

Week 1 (6 hours):

- Monday — Deepavali holiday
- Wednesday 1pm–5pm: Lecture on Topic 1 and Practical 1
- Thursday 11am–1pm: Practical 3

Week 2 (8 hours):

- Monday 3pm–5pm: Topic 2
- Wednesday 1pm–5pm: Topic 2 + Practical 2
- Thursday 11am–1pm: Practical 2

Week 3 (2 + 6 hours):

- Monday 3pm–5pm: **Test** (17%. Q1: CO2, Topic 2; Q2: Topic 1, Topic 3)
- Wednesday 1pm–5pm: Practical 4 by Dr Goh YK. Announcement(?) of **Assignment** (17%)
- Thursday 11am–1pm: Topic 3 and Topic 4 by Dr Goh YK

Week 4 (8 hours): To be arranged by Dr Goh YK

Week 5 (2 + 6 = 8 hours): To be arranged by Dr Goh YK and Dr Yong CK

Week 6 (8 hours): To be arranged by Dr Yong CK (Quiz 1 — 8%??)

Week 7 (2 hours + extra revision?): To be arranged by Dr Yong CK (Quiz 2 — 8%??)

Total course hours = 48 hours

Coursework: Test (20%) + Assignment (20%) + Quizzes (20%)

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

§1.2 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.

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

The knowledge acquired from this subject will be useful in the following UTAR subjects which require a lot of computation with numbers:

- UECM2623 Numerical Methods and Statistics
- UECM2483 Statistical Simulation for Insurance and Finance: The array manipulation in our Topic 2 can be applied to UECM2483 Topic 2. However, the lecturer handling this subject prefers Excel and R.
- UECM3034 Numerical Methods (another important programming language Fortran will be used and Fortran program can be called from Python)
- UECM3764 Computational Finance: Python or C++ will be used depending on the lecturer. The scientific programming skill is required in Topics 2 to 5.
- UECM3543 Case Study on Investment and Trading: A lot of Python is used from electronic trading system to technical analysis!
- UECM3553 Financial Modelling
- UECM3993 Predictive Modelling
- UECS2053/UECS2153/UEMH3163 Artificial Intelligence
- UECS3213/UECS3453 Data Mining
- UEMH4113 Automation and Robotics
- UEEP2083 Computational Physics: This subject is conducted with the expensive commercial software MATLAB. What we cover in Topic 3 is equivalent to UEEP2083 Topic 1. UEEP2083 Topic 2 ODE and Topic 3 PDE are covered in UECM3033. UEEP2083 Topic 4 (random systems) is covered in UECM2483 Statistical Simulation for Insurance and Finance (the name may be changed to Statistical Simulation in future). The references Giordano and Nakanishi [2006], Woolfson and Pert [1999] contain applications of "scientific computing" to physics but they are far too advanced for year 1 and year 2 students.
- UEMX4293 Finite Element Method in Structural Engineering
- UEMK4333 Process Optimisation and Simulation: Defining function and calling standard optimisation functions require the knowledge from Topic 3. However, optimisation normally used more specialised software.
- UEME4343 Computational Methods in Solid Mechanics
- UEME4213 Computational Fluid Dynamics: This subject also uses specialised commercial software ANSYS Fluent (GAMBIT meshing software was used before Fluent v6.3. It was discontinued in new Fluent updates with ANSYS Meshing).

§1.3 Reference Books and Academic Journals for Scientific Computing

- Python-based books:
  - Jaan Kiusalaas, Numerical Methods in Engineering with Python 3, 3rd edition, Cambridge University Press 2013 (Older version: Kiusalaas [2005])
  - Hans Petter Langtangen, A Primer on Scientific Programming with Python, Springer 2016 (Older version: Langtangen [2009])
  - Langtangen [2004]
  - Haenel et al. [2015]
  - Varoquaux et al. [2017] from `https://www.scipy-lectures.org/`.
  - Vaingast [2009]
  - Hilpisch [2015]
  - Swamynathan [2017]
- Intermediate level books:
  - Tveito et al. [2010]
  - Giordano et al. [2009]
- Advanced scientific books:
  - Danaila et al. [2007],
  - Quarteroni and Saleri [2006],
  - Heath [1997],
  - Deuflhard and Hohmann [2003],
  - Blanchet and Charbit [2006],
  - Beers [2006],
  - Holzbecher [2007],
  - Danaila et al. [2007],
  - Elnashaie and Uhlig [2007],
- Books on Applications:
  - Computational Mathematics Driven by Industrial Problems, 2009

Scientific computing related journals listed in Web of Science:

- 1521-9615, COMPUTING IN SCIENCE & ENGINEERING, Q3
- 2192-1962, Human-centric Computing and Information Sciences, Q1
- 1530-9827, JOURNAL OF COMPUTING AND INFORMATION SCIENCE IN ENGINEERING, Q3
- 0885-7474, JOURNAL OF SCIENTIFIC COMPUTING, Q1
- 1064-8275, SIAM JOURNAL ON SCIENTIFIC COMPUTING, Q1
- 0168-9274, APPLIED NUMERICAL MATHEMATICS,Q1,2021
- 0006-3835, BIT NUMERICAL MATHEMATICS,Q2
- 0764-583X, ESAIM-MATHEMATICAL MODELLING AND NUMERICAL ANALYSIS-MODELISATION MATHEMATIQUE ET ANALYSE NUMERIQUE,Q2
- 1570-2820, Journal of Numerical Mathematics,Q1
- 1004-8979, Numerical Mathematics-Theory Methods and Applications,Q2
- 0029-599X, NUMERISCHE MATHEMATIK,Q1

- 0927-6467, RUSSIAN JOURNAL OF NUMERICAL ANALYSIS AND MATHEMATICAL MODELLING,Q3
- 1335-9150, COMPUTING AND INFORMATICS,Q4
- 2192-1962, Human-centric Computing and Information Sciences,Q1
- 1079-8587, INTELLIGENT AUTOMATION AND SOFT COMPUTING,Q3
- 1598-5865, Journal of Applied Mathematics and Computing,Q1
- 1530-9827, JOURNAL OF COMPUTING AND INFORMATION SCIENCE IN ENGINEERING,Q3
- 2210-5379, Sustainable Computing-Informatics & Systems,Q1

§1.4 Software for Scientific Computing

- Python + Numpy + Scipy + Matplotlib + Pandas
  - Spyder: all-in-one IDE. Like MATLAB
  - Jupyter Notebook (popular but I dislike it)
  - Text Editor + Python Shell (e.g. Anaconda Prompt): A direct way to communicate with Python and can learn how to compare files (`https://www.youtube.com/watch?v=QKBcHuA3VJE`)
- MATLAB or Octave (`https://www.gnu.org/software/octave/`
- Scilab [Campbell, 2006]
- R (`https://www.r-project.org/`)
- Julia (`https://julialang.org/`)

In a **Python shell** working environment, we need to be familiar with the following terms:

- Working directory: The default directory for us to read and write a file.
- Session management: Python shell does 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:

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

  In Jupyter, one can use `%hist -f my_history.py` to store the commands in a session.
- Help and Documentation: Python is popular because it has a very good library with a detail documents.
- Editor: Anaconda recommends VS Code but I don't think the computer lab install VS Code, so Spyder may be a good editor and environment for Python beginners.

Applications of scientific computing:

- data analysis: Approximating 'data' using statistical models Grus [2015].
- machine learning: building 'intelligent model' from data for marketing, information management and business decision making [Richert and Coelho, 2013].
- Engineering modelling and calculations.

§1.5 Variables, Working Environments and Getting Help

Variables are **names** (Start with a capital or small letter and then letters and/or numbers). It is used to **refer to values/data/objects in computer**.

Example 1.5.1.

- `Variable1 = 1 + 2 + 3.5`
- `variable2 = "A string" + "another string"*2`
- `var_3 = sin`
- `Var_3 = var_3(pi)`

Working Environments:

- Start a session: `python`
- End a session: `quit()`
- Load a module or package or function: `import modulename (as abbrev)`, `from math import sin, cos, pi`
- Reload a module: `importlib.reload(modulename)`
- Unload a package or module: `del modulename`
- Working directory in current session: `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)`

Online help:

- Online help: `https://docs.python.org/3/`, `https://www.python.org/about/help/`
- Local help: `help(sin)`
- Specific help: help([object]) → If the argument is a string, then the string is looked up as the name of a module, function, class, method, keyword, or documentation topic, and a help page is printed on the console. If the argument is any other kind of object, a help page on the object is generated.
- Numpy specific help: `np.info(np.sin)`, `np.lookfor('create array')`

§1.6 Basic Data Types

The basic data types in Python are limited:

- 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)`

Question: Is memorising necessary?

Answer: Yes! The most commonly used data types and operations must be memorised — even if we memorise incorrectly, we can use keywords to find from programming documentations ....

§1.6.1 Characters, Strings, Texts

For a binary computer, the basic data is usually in the 8-bit byte and the multiples of 8-bit bytes. The following illustrates that for the same binary number, it can represent a **character** or an **integer** depending on how we specific the **type**.

Binary	As ASCII	As Integer
0010 1100	`.` ← `chr(int('00101100',2))`	0b00101100 = 44
0010 1110	`,` ← `chr(int('00101110',2))`	0b00101110 = 46
0011 0000	`0` ← `chr(int('00110000',2))`	0b00110000 = 48
0011 1001	`9` ← `chr(int('00111001',2))`	0b00111001 = 57
0100 0001	`A` ← `chr(int('01000001',2))`	0b01000001 = 65
0101 1010	`Z` ← `chr(int('01011010',2))`	0b01011010 = 90
0110 0001	`a` ← `chr(int('01100001',2))`	0b01100001 = 97

In reality, computers are more complicated because of the ordering of multiple bytes can either be "Little Endian" or "Big Endian" depending on the type of CPU.

A **string** is an ordered collection of **characters**.

In Python language, it is usually expressed as things in **double quotes** or **single quotes** with possible **control/escape character** \. Two important control characters are \n and \t.

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, use `print('no newline', end='')` if new line is not desired)
- English/Latin related functions: `.lower()`, `.upper()`, `.capitalize()`
- Computer representations: `ord()`, `chr()`

§1.6.2 Booleans

The simplest logic is Boolean logic, containing `True` and `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

§1.6.3 Integers

Python Integers = Integers of any size limited by computer memory. E.g. `-123`, `987_654_321`

Very large integers can used up all computer 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 $2^{63}-1$).

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` : floating-point number division
- `3 == 4`, `3 != 4` : Check for 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
- `i1 = int("123", 4)`: convert a string to an integer (in the case, as a base 4 integer)
- Format to strings: `str()` (decimal string), `bin()` (binary string), `oct()` (octal string), `hex()` (hexadecimal string), `"%6d" % 7` (C-style formatting), `":6d".format(7)` (C#-style formatting), etc.
- Convert to and from Boolean: `bool(i1)`, `int(b1)`

§1.6.4 Floating Point Numbers, Arithmetic and Relations

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.

Expression in Python:

- Decimal/positional notation: 1.3, .1, 15., 1_2345.67
- Scientific notation: 1e-2, 3.5e2, -0.1e2, ...
- Engineering notation (the exponent is always a multiple of 3, related to the **unit prefix**, tera 10^{12}, giga 10^9, mega 10^6, kilo 10^3, milli 10^{-3}, micro 10^{-6}, nano 10^{-9}, pico 10^{-12}): 405.77e-3, 23e9, ...

Operations related to floating-point numbers:

- Special constants: `math.e`, `math.pi`, `math.tau`, `math.inf`, `math.nan`
- `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` : Be careful with floating-point equality: 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)
- `float("123.456e-1")`: convert a string to a floating point number
- Format to strings: `str()`
- Real function library: `import math`

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)`
- `abs()`, $-x$, +, -, *, /, **, `.conjugate()`, `.imag`, `.real`, ==, !=
- Complex function library: `import cmath`

§1.6.5 Formatting Numbers

Python provides

- C-style formatting controls for numbers: `%f`, `%g` (for shortest representation), `%e` (scientific notation)
- C#-style formatting controls for numbers: `"{:6.3f}".format(1/3)` — Occupying 6 characters and 3 decimal places in this case with spaces.

Example 1.6.1. Express the fine structure constant $\alpha = 7.297352568 \times 10^{-3}$ in Python and print out its value in 6 decimal places and in scientific notation with 6 significant figures.

§1.6.6 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.

1. *Tuple*. It is constructed using curved brackets (e.g. `(1,1.0,"Helo")`) or `tuple()` for programming construction and has virtually no methods associated with it.
2. *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`.

Python tuples (limited used):

- Use for value matching: `x1,x2,x3 = 3,4,5`

  Note: Somethings the round brackets can be ignored
- Swapping values: `x, y = y, x`

Python lists:

- 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.
- A list of list of numbers can be used to represent matrix but it is slow (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 B x A 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]]
]
```

§1.7 Functions

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

The reasons for defining functions include

(a) Do not repeat code!

(b) Give it a reasonable name so that it is easy to know its purpose.

On point (a), let us investigate an example below which consists of many repeated statements.

```
1  a_norm = 0.0
2  for i in range(0,100):
3      a_norm += a[i]*a[i]    # First occurrence
4  ...
5  b_norm = 0.0
6  for i in range(0,100):
7      b_norm += b[i]*b[i]    # Repetition
8
9  ...
10 c_norm = 0.0
11 for i in range(0,100):
12     c_norm += c[i]*c[i]    # Repetition
```

We can see that lines 1 to 3, lines 5 to 7 and lines 10 to 12 are repeating and by defining a function, we can structure the program to a program below.

```
def norm(v):          # Define a function. Single instance of the code
    v_norm = 0.0
    for i in range(0,100):
        v_norm += v[i]*v[i]
    return v_norm
...
a_norm = norm(a)      # Calling the function
...
b_norm = norm(b)      # Calling the function
...
c_norm = norm(c)      # Calling the function
```

On the point (b), anyone who knows linear algebra well would be able to recognise the purpose of `scipy.linalg.norm` in measuring length and distance.

A function can be abstractlly represented as

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

However, for simple functions, it is possible to use the following forms

- A simple Python function can sometimes be expressed as a **single-line def**:

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

  or using **lambda expression**:

  ```
  f = lambda x: an_expression_of x ...
  ```

Example 1.7.1. Solve the following nonlinear equation using Python

$$\sin(2x) = 2\cos(x). \qquad \text{(3 marks)}$$

Sample Solution

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

# Use the algorithm from Scipy to estimate out an solution
import scipy.optimize
estimate_soln = scipy.optimize.fsolve(f,0.0)
print("The solution (close to 0) is", estimate_soln)
```

Example 1.7.2 (UEEP2083 Computational Physics Final Exam Sept 2015, Q1(c)(iii) with modification)**.** Create the Python commands to solve integral $\int_0^{10} (x^{x/2} + 0.9x)dx$. (2 marks)
[Remark: Similar to Example 1.7.1, for this example, you need to define a function and pass it to a "numerical quadrature" (which be studied in UECM3034) using `scipy.integrate.quad`]

Example 1.7.3 (Final Exam Sept 2012, Q2(b) with modification)**.** Write a Python function that converts a temperature in Fahrenheit to a temperature in Celcius according to the following equation

$$[^\circ C] = \frac{5([^\circ F] - 32)}{9}.$$

Show an example of using the function `f2c` in the Python command window. (8 marks)

Some basic functions are already defined in Python (by importing appropriate modules):

- Getting the minimum and maximum of $x_1, x_2, ..., x_n$: `min(`$x_1, x_2, ..., x_n$`)`, `max(`$x_1, x_2, ..., x_n$`)`
- Integer functions: `gcd(x1,...,xn` (Greatest common divisor of n integers)
- Convert base-`n` string x to integer: `int(`x`,n)`
- Convert integer x to binary, octal and hexademical string representation: bin(x), oct(x), hex(x)
- Euclidean geometry: `radians(degree)`, `degrees(radian)`
- Functions to convert floating-point numbers to integers: textttround(x), `ceil(x)`, `floor(x)`
- Functions from Calculus: `exp(x)`, `log(x)` (Natural logarithmic), `log10(x)` (base-10 logarithmic), `pow(x,y)` (x^y), `sin(x)`, `cos(x)`, `tan(x)`, `asin(x)`, `acos(x)`, `atan(x)`
- Function to test floating-point numbers: `isfinite(x)`, `isinf(x)`, `isnan(x)`

More linear algebra and scientific functions are available in the Scipy's linalg modules.

Example 1.7.4. Write down the Python command to calculate $\sin 10^\circ \sin 30^\circ \sin 50^\circ \sin 70^\circ$.

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))
```

Exercise: What is the alternative way to calculate with Python without using the `radians()` function?

Example 1.7.5. Write down the Python command to calculate $\tan 6^\circ \tan 42^\circ \tan 66^\circ \tan 78^\circ$.

Example 1.7.6. Write down Python command to calculate $\cos 42^\circ \cos 18^\circ - \sin 42^\circ \sin 18^\circ$. Are you able to find the value exact using trigonometric equality?

§1.7.1 Positional-only parameters

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.

Positional Parameters Example

```
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)
```

§1.7.2 Variable Arguments

When a function or a method of the same name has different number of inputs, it is said to have variable arguments. In Python, this can be achieved by three syntax, one syntax is to use the **default arguments**:

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

another syntax is to use the **variable length argument lists**:

```
def bfunction(farg, *args):
    ...
    for i in args: do something to the value i
```

and one syntax is to use the **keyword-ed argument lists** (which uses the Dictionary data structure mentioned in Section 2.1 in Topic 2):

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

An example for **default arguments** is `np.max(A, axis=None).` or `A.max(axis=None)`. Consider the following 2-D array:

$$\mathtt{A} = \begin{array}{|c|c|c|c|c|} \hline -3 & -4 & -7 & 8 & -9 \\ \hline 2 & 1 & -2 & 5 & -5 \\ \hline \end{array}$$

Then

$$\texttt{np.max(A)} \text{ or } \texttt{A.max()} \Rightarrow 8$$

but

$$\texttt{np.max(A,1)} \text{ or } \texttt{A.max(1)} \Rightarrow \texttt{array([8, 5])}.$$

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

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

Apart from `print`, I don't see this syntax often.

As for the keyword-ed argument lists, we can see how it is used in a real world situation (e.g. plotting financial data) illustrated by the following Python program (taken from `https://realpython.com/numpy-array-programming/#what-is-vectorization`)

```
# https://realpython.com/numpy-array-programming/#what-is-vectorization
# https://matplotlib.org/1.3.1/examples/pylab_examples/filledmarker_demo.html
prices = np.full(100, np.nan)
prices[[0, 25, 60, -1]] = [80., 30., 75., 50.]
x = np.arange(prices.size)
is_valid = ~np.isnan(prices)
prices = np.interp(x=x, xp=x[is_valid], fp=prices[is_valid])
prices += np.random.randn(prices.size) * 2

fig = plt.figure()
fig.show()
ax = fig.add_subplot(1,1,1)
kwargs = {'marker':'o', 'markersize':12, 'linestyle':''}
ax.plot(prices)
ax.set_title('Price History')
ax.set_xlabel('Time')
ax.set_ylabel('Price')
mn = np.argmin(prices)
mx = mn+np.argmax(prices[mn:])  # By low sell high
ax.plot(mn, prices[mn], color='green', **kwargs)
ax.plot(mx, prices[mx], color='red', **kwargs)
plt.legend(loc=2)
plt.draw()
```

§1.8 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:

```
1 width = 70
2 pre = "\n"*2 + "*"*width + "\n*" + " "*(width-2) + "*\n*"
3 post = "*\n*" + " "*(width-2) + "*\n" + "*"*width
4 greet = "\n\nEnter your name: "
5 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()`

§1.9 Script Files

Python **script**:

- A file which can be opened using **notepad** and we can read the content of the file to be a Python program.
- It 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.
- The file ends with `.py`.

Jupyter notebook for Python:

- A JSON file which need Jupyter notebook to read. Open using notepad shows us something different from Python program.
- It needs to be opened by a Browser and commands will be send to Python shell through Browser.
- The file ends with `.ipynb`.

Warning: Do not save Python program in Word format. Word is more complicated format than Jupyter notebook.

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

A comment starts with #.

For a long / multi-paragraph comments, we usually treat them as (doc-)strings which open and end with either `"""` or `'''`.

Example 1.9.1 (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

$$\begin{aligned} a^2 &= b^2 + c^2 - 2bc\cos A, \\ b^2 &= a^2 + c^2 - 2ac\cos B, \\ c^2 &= a^2 + b^2 - 2ab\cos C. \end{aligned}$$

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**. (10 marks)

Sample Solution

```
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 1.9.2. Write a Python script to generate the numeric tables for functions $\sin x$ ($0 \le x \le \pi$), $\cos x$ ($0 \le x \le \pi$), $\exp x$ ($-2 \le x \le 2$) and $\ln x$ ($10^{-2} \le x \le 10^2$) using the precision 10^{-2} (i.e. up to two decimal places).

It is probably straightforward to use the for loop. We can also use while loop but it involves a little bit more steps. The simplest answer is used array which is discussed in the next topic.

```
from math import sin, cos, exp, log, pi

#
# Sine and cosine has common interval (0, pi)
# We calculate how many points we need for building table
#
N1 = int((pi-0)/0.01)+1
```

```
#
# We start from 0 and increase by 0.01 in the for loop
#
x = 0.0
print("x".center(5), "sin(x)".center(8), "cos(x)".center(8))
print("{:5.2f}  {:-7.4f}  {:-7.4f}".format(x, sin(x), cos(x)))
for i in range(N1):
    x  = x+0.01
    y1 = sin(x)
    y2 = cos(x)
    print("{:5.2f}  {:-7.4f}  {:-7.4f}".format(x, y1, y2))

#
# Leave a blank line between tables
#
print()

#
# We now create the numeric table for exp(x), -2 <= x <= 2
# It is similar to sine and cosine except that we start from -2
# instead of 0.
#
N2 = int((2-(-2))/0.01)+1
x = -2.0
print("x".center(5), "exp(x)".center(8))
print("{:5.2f}  {:-7.4f}".format(x, exp(x)))
for i in range(N2):
    x  = x+0.01
    y1 = exp(x)
    print("{:5.2f}  {:-7.4f}".format(x, y1))

#
# Leave a blank line between tables
#
print()

#
# We now create the numeric table for log(x), 0.01 <= x <= 100
# It is similar to exp(x) except that we start from 0.01
# instead of -2.
#
N3 = int((100-0.01)/0.01)+1
x = 0.01
print("x".center(5), "log(x)".center(8))
print("{:5.2f}  {:-7.4f}".format(x, log(x)))
for i in range(N3):
    x  = x+0.01
    y1 = log(x)
    print("{:5.2f}  {:-7.4f}".format(x, y1))
```

§1.10 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.

1. **If only**:

```
if condition:
    do_something_block
```

2. **If-else statement**:

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

3. **One-line form (ternary operator)**:

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

4. **Multiple conditions**: A sample syntax is given below

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

Example 1.10.1 (If-else statement and One-line form)**.** The one-line form is very convenient for short and 'complement' statements. For example, for a step function:

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

If-else statement

```
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 1.10.2 (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
```

(17 marks)

Multi-condition statement

```python
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 1.10.3. Given a cubic equation

$$a_0x^3 + a_1x^2 + a_2x + a_3 = 0 \Rightarrow \boxed{x^3 + ax^2 + bx + c = 0.} \tag{1.1}$$

Let $\boxed{x = t - \dfrac{b}{3a}}$ 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 $h = 0$) $\Rightarrow q = 0 \Rightarrow x_1 = x_2 = x_3 = -\frac{b}{3a}$
- $p \neq 0 \Rightarrow t_1 = \frac{3q}{p}$, $t_2 = t_3 = -\frac{3q}{2p} \Rightarrow x_k = t_k - \frac{b}{3a}$, $k = 1, 2, 3$.

- 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:

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

Sample Implementation of Cubic Equation Solver

```
def quadratic(poly):
    if len(poly) == 2:
        a, b = poly # Linear equation
        return -b/a
    elif len(poly) == 3:
        a, b, c = poly
    else:
        print("Coefficients", poly, "are not quadratic")
        return None
    # https://en.wikipedia.org/wiki/Quadratic_equation
    r = b ** 2 - 4*a*c
    if r < 0:
        from cmath import sqrt
    else:
        from math import sqrt
    sr = sqrt(r)
    return ((-b + sr) / (2*a), (-b - sr) / (2*a))

print(quadratic([1,2,2]))     # No real solution
print(quadratic([1,2,1]))     # x1 = x2 = -1
print(quadratic([1,3,2]))     # x1 = -1, x2 = -2

def cubic(poly):
    if len(poly) == 3:
        return quadratic(poly)
    elif 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 = t1 - b_3a
    x2 = t2 - b_3a
    x3 = 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

# Multiple roots:
print(cubic([-1, 0, 0, 0]))      # -x^3
print(cubic([1,-9,27,-27]))      # (x-3)^3

# 3 distinct roots:
print(cubic([1, 6, 11, 6]))      # (x+1)*(x+2)*(x+3)
print(cubic([2, 25, 82, 35]))    # (2*x+1)*(x+5)*(x+7)

# 2 complex roots:
print(cubic([1, -1, 1, -1]))     # (x^2+1)*(x-1)
print(cubic([2, 1, 2, 1]))
```

Example 1.10.4. A quintic equation

$$a_0x^4 + a_1x^3 + a_2x^2 + a_3x + a_4 = 0 \Rightarrow x^4 + ax^3 + bx^2 + cx + d = 0 \tag{1.2}$$

has the solutions

$$x = -\frac{a}{4} + \frac{\pm_s\sqrt{\alpha+2y} \pm_t \sqrt{-\left(3\alpha + 2y \pm_s \frac{2\beta}{\sqrt{\alpha+2y}}\right)}}{2}.$$

$$\text{where } \begin{cases} \alpha = b - \dfrac{3a^2}{8}, \quad \beta = c - \dfrac{ab}{2} + \dfrac{a^3}{8}, \quad \gamma = d - \dfrac{ac}{4} + \dfrac{a^2b}{16} - \dfrac{3a^4}{256}, \\ P = -\dfrac{\alpha^2}{12} - \gamma, \quad Q = -\dfrac{\alpha^3}{108} + \dfrac{\alpha\gamma}{3} - \dfrac{\beta^2}{8}, \quad R = -\dfrac{Q}{2} \pm \sqrt{\dfrac{Q^2}{4} + \dfrac{P^3}{7}}, \quad U = \sqrt[3]{R}, \\ y = -\dfrac{5}{6}\alpha + \begin{cases} U = 0 & \to -\sqrt[3]{Q} \\ U \neq 0, & \to U - \frac{P}{3U}, \end{cases} \end{cases}$$

Sample Implementation of Quintic Equation Solver

```
def quartic(poly):
    a0, a1, a2, a3, a4 = poly
    a, b, c, d = a1/a0, a2/a0, a3/a0, a4/a0

    # Ferrari's method (https://en.wikipedia.org/wiki/Quartic_equation)
    alpha = b - 3*a**2 / 8
    beta  = c - a*b / 2 + a**3 / 8
    gamma = d - a*c / 4 + a**2 * b / 16 - 3*a**4 / 256

    from cmath import sqrt
    if beta == 0:
        x1 = -a/4 + sqrt((-alpha + sqrt(alpha**2-4*gamma))/2)
        x2 = -a/4 - sqrt((-alpha + sqrt(alpha**2-4*gamma))/2)
        x3 = -a/4 + sqrt((-alpha - sqrt(alpha**2-4*gamma))/2)
        x4 = -a/4 - sqrt((-alpha - sqrt(alpha**2-4*gamma))/2)
    else:
        P = -alpha**2/12 - gamma
        Q = -alpha**3/108 + alpha*gamma/3 - beta**2/8
        R = -Q/2 + sqrt(Q**2/4 + P**3/27)
        U = R**(1/3)
        if U == 0.0:
            y = -5/6*alpha - Q**(1/3)
        else:
            y = -5/6*alpha + U - P/(3*U)
        W = sqrt(alpha + 2*y)
        D = sqrt(-(3*alpha+2*y+2*beta/W))
        E = sqrt(-(3*alpha+2*y-2*beta/W))
        x1 = ( W + D)/2 - a/4
        x2 = ( W - D)/2 - a/4
        x3 = (-W + E)/2 - a/4
        x4 = (-W - E)/2 - a/4

    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
    x4 = x4 if abs(x4.imag) > 1e-16 else x4.real

    return x1, x2, x3, x4

print(quartic([1, 0, 2, 0, 1]))      # (x^2+1)^2
print(quartic([1, 0, 0, 0, -1]))     # (x^2+1)(x^2-1)
print(quartic([1, 10, 35, 50, 24]))  # (x+1)(x+2)(x+3)(x+4)
print(quartic([1, -9,27,-31,12]))    # (x-1)^2(x-3)(x-4)
print(quartic([1, -5, 0, 10, 24]))    # (x^2+2x+2)(x-3)(x-4)
```

§1.11 Loops and Recursion

When we want to solve a "repeating" problem, for example, if there are thirty quadratic equations with different sets (a, b, c) in Example 1.10.2, are we going to write a script with 30 lines as follows?

```
solve_quadratic_equation(1,2,3)
solve_quadratic_equation(4,5,6)
... another 27 lines ...
solve_quadratic_equation(5,-2,7)
```

Or is it possible two write a script with less lines such as?

```
all_sets_of_values = [(1,2,3), (4,5,6),
    ... another 27 sets of (a,b,c), (5,-2,7)]  # Takes 5 or 6 lines?
go through a,b,c in all_sets_of_values
    solve_quadratic_equation(a,b,c)
```

The "go through all cases" statements or looping statements are called "loops". Python provides two kinds of "loop" — for loop and while loop as well as recursion (Section 1.11.4) to support looping statements.

§1.11.1 For Loop

for loop statement

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

Example 1.11.1 (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 $k = 100$. (7 marks)

Sample Solution

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

def S(k):
    # f(1) + f(2) + ... + f(k)  # k+1 is not included in Python
    return sum(f(n) for n in range(1,k+1))

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

Example 1.11.2 (Final Exam Sept 2014, Q2(c) with modification)**.** Write a Python code to calculate the following summation

$$3(2+1)+4(3+2+1)+5(4+3+2+1)+6(5+\cdots+1)+\cdots+1000(999+\cdots+1). \quad \text{(10 marks)}$$

[Hint: Python's range, list comprehension and sum]

Sample Solution

```
# Each term in the sum is
def f(n):
    return n*sum(range(1,n))  # n (1 + ... + n-1)

# The summation is f(3) + f(4) + ... + f(1000)
sum(f(i) for i in range(3,1001))
```

Example 1.11.3 (Final Exam Sept 2015, Q4(b)). A weighted mean is used when there are varying weights for the data values. For a data set given by $x = \{x_1, x_2, x_3, \cdots, x_n\}$ and corresponding weights for each x_i, $w = \{w_1, w_2, w_3, \cdots, w_n\}$, 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. (8 marks)

Sample Solution using For Loop

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

Example 1.11.4 (Final Exam Oct 2019 Q1(b)). *Horner's method* (`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 $a_0, a_1, \cdots, a_n$ of the polynomial $p(x)$ as `coeffs` and a value `x` and then returns the value of the polynomial $p(x)$ at `x`.

Example 1.11.5 (Discrete Mathematics, Tutorial 1, Q1(a)(iv)). Generate the truth table of the statement $\sim s \to (p \wedge (q \vee r))$ 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 1.11.6 (Number Theory. Final Exam Oct 2019 Q1(a)). Study the following Python script which generates a multiplication table for simple finite field and write down the output:

```python
def galois_table(prime_number):
    print("   |",end="")
    for i in range(prime_number):
        print(f"{i:5d}",end="")
    print("\n" + "-"*(3+5*prime_number))
    for i in range(prime_number):
        print(f"{i} |", end="")
        for j in range(prime_number):
            v = i*j % prime_number
            print(f"{v:5d}",end="")
        print()

galois_table(5)
print("="*70)
galois_table(7)
```

Solution: You need to learn how the following tables are generated.

```
  |    0    1    2    3    4
----------------------------
0 |    0    0    0    0    0
1 |    0    1    2    3    4
2 |    0    2    4    1    3
3 |    0    3    1    4    2
4 |    0    4    3    2    1
======================================================================
  |    0    1    2    3    4    5    6
--------------------------------------
0 |    0    0    0    0    0    0    0
1 |    0    1    2    3    4    5    6
2 |    0    2    4    6    1    3    5
3 |    0    3    6    2    5    1    4
4 |    0    4    1    5    2    6    3
5 |    0    5    3    1    6    4    2
6 |    0    6    5    4    3    2    1
```

Example 1.11.7 (Probability and Statistics)**.** Use Python's `scipy.stats.norm.cdf` function to generate the following cumulative normal distribution table.

```
  z  |  0.00|  0.01|  0.02|  0.03|  0.04|  0.05|  0.06|  0.07|  0.08|  0.09
-----+------+------+------+------+------+------+------+------+------+------
 0.00|0.5000|0.5040|0.5080|0.5120|0.5160|0.5199|0.5239|0.5279|0.5319|0.5359
 0.10|0.5398|0.5438|0.5478|0.5517|0.5557|0.5596|0.5636|0.5675|0.5714|0.5753
 0.20|0.5793|0.5832|0.5871|0.5910|0.5948|0.5987|0.6026|0.6064|0.6103|0.6141
 0.30|0.6179|0.6217|0.6255|0.6293|0.6331|0.6368|0.6406|0.6443|0.6480|0.6517
 0.40|0.6554|0.6591|0.6628|0.6664|0.6700|0.6736|0.6772|0.6808|0.6844|0.6879
 0.50|0.6915|0.6950|0.6985|0.7019|0.7054|0.7088|0.7123|0.7157|0.7190|0.7224
 0.60|0.7257|0.7291|0.7324|0.7357|0.7389|0.7422|0.7454|0.7486|0.7517|0.7549
 0.70|0.7580|0.7611|0.7642|0.7673|0.7704|0.7734|0.7764|0.7794|0.7823|0.7852
 0.80|0.7881|0.7910|0.7939|0.7967|0.7995|0.8023|0.8051|0.8078|0.8106|0.8133
 0.90|0.8159|0.8186|0.8212|0.8238|0.8264|0.8289|0.8315|0.8340|0.8365|0.8389
 1.00|0.8413|0.8438|0.8461|0.8485|0.8508|0.8531|0.8554|0.8577|0.8599|0.8621
```

Sample Solution

```python
from scipy import stats

L=2; N=6; D=N-2
z_max = 10 # 37
print("z".center(L+3),end="")
for i in range(10):
```

```
    print(f"|{0.01*i:{N}.2f}",end="")
print("\n"+"-"*(L+3)+("+"+"-"*N)*10)
for i in range(z_max+1):
    print(f"{0.1*i:{L+3}.{L}f}",end="")
    z = 0.1*i
    print(f"|{stats.norm.cdf(z):{N}.{D}f}",end="")
    for j in range(1,10):
        z = 0.1*i + 0.01*j
        print(f"|{stats.norm.cdf(z):{N}.{D}f}",end="")
    print()
```

Exercise: Can you modify it to generate a table with z varying from 0 to 3.49?

Example 1.11.8 (Probability and Statistics)**.** Use Python's `scipy.stats.t.ppf` function to generate the critical values of the t distribution.

	Area to the Right (t-distribution)					
df	0.05	0.025	0.01	0.005	0.001	0.0005
1	6.314	12.706	31.821	63.657	318.309	636.619
2	2.920	4.303	6.965	9.925	22.327	31.599
3	2.353	3.182	4.541	5.841	10.215	12.924
4	2.132	2.776	3.747	4.604	7.173	8.610
5	2.015	2.571	3.365	4.032	5.893	6.869
6	1.943	2.447	3.143	3.707	5.208	5.959
7	1.895	2.365	2.998	3.499	4.785	5.408
8	1.860	2.306	2.896	3.355	4.501	5.041
9	1.833	2.262	2.821	3.250	4.297	4.781
10	1.812	2.228	2.764	3.169	4.144	4.587
11	1.796	2.201	2.718	3.106	4.025	4.437
12	1.782	2.179	2.681	3.055	3.930	4.318
13	1.771	2.160	2.650	3.012	3.852	4.221
14	1.761	2.145	2.624	2.977	3.787	4.140
15	1.753	2.131	2.602	2.947	3.733	4.073
16	1.746	2.120	2.583	2.921	3.686	4.015
17	1.740	2.110	2.567	2.898	3.646	3.965
18	1.734	2.101	2.552	2.878	3.610	3.922
19	1.729	2.093	2.539	2.861	3.579	3.883
20	1.725	2.086	2.528	2.845	3.552	3.850
21	1.721	2.080	2.518	2.831	3.527	3.819
22	1.717	2.074	2.508	2.819	3.505	3.792
23	1.714	2.069	2.500	2.807	3.485	3.768
24	1.711	2.064	2.492	2.797	3.467	3.745
25	1.708	2.060	2.485	2.787	3.450	3.725
26	1.706	2.056	2.479	2.779	3.435	3.707
27	1.703	2.052	2.473	2.771	3.421	3.690
28	1.701	2.048	2.467	2.763	3.408	3.674
29	1.699	2.045	2.462	2.756	3.396	3.659
30	1.697	2.042	2.457	2.750	3.385	3.646
32	1.694	2.037	2.449	2.738	3.365	3.622
34	1.691	2.032	2.441	2.728	3.348	3.601
36	1.688	2.028	2.434	2.719	3.333	3.582
38	1.686	2.024	2.429	2.712	3.319	3.566
40	1.684	2.021	2.423	2.704	3.307	3.551
42	1.682	2.018	2.418	2.698	3.296	3.538
44	1.680	2.015	2.414	2.692	3.286	3.526
46	1.679	2.013	2.410	2.687	3.277	3.515
48	1.677	2.011	2.407	2.682	3.269	3.505
50	1.676	2.009	2.403	2.678	3.261	3.496
60	1.671	2.000	2.390	2.660	3.232	3.460
70	1.667	1.994	2.381	2.648	3.211	3.435
80	1.664	1.990	2.374	2.639	3.195	3.416
90	1.662	1.987	2.368	2.632	3.183	3.402

100	1.660	1.984	2.364	2.626	3.174	3.390
120	1.658	1.980	2.358	2.617	3.160	3.373
150	1.655	1.976	2.351	2.609	3.145	3.357
200	1.653	1.972	2.345	2.601	3.131	3.340
300	1.650	1.968	2.339	2.592	3.118	3.323
400	1.649	1.966	2.336	2.588	3.111	3.315
500	1.648	1.965	2.334	2.586	3.107	3.310
600	1.647	1.964	2.333	2.584	3.104	3.307
oo	1.645	1.960	2.326	2.576	3.090	3.291

Example 1.11.9 (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:

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

If $x_{10} = 10$, it is represented as '`X`'. Write a Python **program script** to implement the function `checkdigit10(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. (5 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'))
```

- Correct definition of function .... [1 mark]

- Proper use of for loop .... [1 mark]
- Demonstrate correct translation of mathematical formula to computer program [3 marks]

Example 1.11.10. 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 $r = \big(10 - \big(x_1 + 3x_2 + x_3 + 3x_4 + x_5 + 3x_6 + x_7 + 3x_8 + x_9 + 3x_{10} + x_{11} + 3x_{12}\big) \bmod 10\big)$. 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-?`.

Example 1.11.11 (Final Exam Oct 2018, Q2(c), CO3)**.** Given a Python function `myst` as follows:

```
def myst(a):
    b = a.copy()
    n = a.size
    for i in range(int(n/2)): b[i],b[n-i-1] = b[n-i-1],b[i]
    return b
```

After you have imported the function `myst` and run `myst(np.array([1,2,3, 5,7,9]))`, what return value will you obtain? Explain in *one sentence* what the function `myst` does? (5 marks)

The next example illustrates how to make "for loop" interesting to beginners by using simple "turtle graphics".

Example 1.11.12. What does the following Python script do?

```
import turtle as tt

# https://www.youtube.com/watch?v=b6AcYxIxXMA
# Hayley Denbraver - Recursion, Fractals, and the Python Turtle Module
```

```
hayley_turtle = tt.Turtle()
hayley_turtle.color("blue")
hayley_turtle.pensize(4)
hayley_turtle.shape("turtle")
hayley_turtle.speed(7)

wn = tt.Screen()
side_length = 200
num_of_trig = 15
angle = 360//num_of_trig
for triangle in range(num_of_trig):
    for i in range(3):
        hayley_turtle.forward(side_length)
        hayley_turtle.right(120)
    hayley_turtle.right(angle)
wn.exitonclick()
```

Solution: The Python script sketch a spiral graph using rotating triangle.

§1.11.2 Early Termination of a Loop

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

The `continue` statement is used to skip some statements in a loop.

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, ...

Application of continue statement: Skip processing certain rows of data, ...

§1.11.3 While Loop

while loop statement

```
while condition:
    block
```

In contrast to "for loop", we seldom use "while loop". However, when we don't know when to stop, a "while loop" statement augmented with `break` statement is preferred and illustrative examples are given below.

Example 1.11.13 (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. (9 marks)

Sample Solution

```
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 1.11.14 (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. (12 marks)

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

```
1 thesum = n = 0
2 while thesum < 200:
3     n = n+1
4     thesum = sum(range(2,2*n+1,2))
5     greatest_n = n-1
6 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 $2 + 4 + ... + 2n < 200$ is 13.

Example 1.11.15 (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.

$$\frac{4}{5} + \frac{5}{6} + \frac{6}{7} + \frac{7}{8} + \frac{8}{9} + \cdots . \qquad \text{(6 marks)}$$

§1.11.4 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 1.11.16 (`https://en.wikipedia.org/wiki/Fibonacci_number`)**.** The mathematical definition of Fibonacci numbers are given by

$$F_0 = 0, \; F_1 = 1, \; F_n = F_{n-1} + F_{n-2}, \quad n = 2, 3, \cdots \tag{1.3}$$

The recursive function implemetation of Fibonacci is shown below.

Solution: Recursive Function Implementation

```
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 1.11.17. Consider a variation of the recurrence relation found in 2017 West Australia Applications Exam (`https://senior-secondary.scsa.wa.edu.au/__data/assets/pdf_file/0006/458997/Mathematics_Applications_Calc_Free_2017.PDF`):

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

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)`, $\cdots$, `T(10)`. Write down the value of `T(4)`.

Example 1.11.18 (Final Exam Sept 2015, Q2(d))**.** The Taylor series of a cosine function at $x = 0$ is

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

Write a recursive function that computes and approximation of cosine. (5 marks)

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)
```

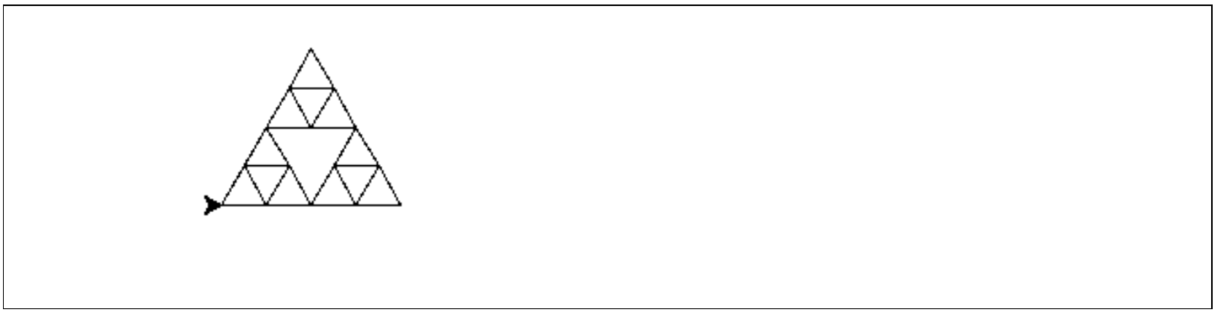
Recursion is a method to perform calculation for programming languages which does not have for loop or while loop.

Recursion is powerful in the generation of very fancy graphics as illustrated below.

Example 1.11.19. Play with the following `sierpinski` function and explain what we can get.

```
# https://stackoverflow.com/questions/25772750/sierpinski-triangle-recursion-using-
import turtle
def draw_sierpinski(length,depth):
```

```
    if depth==0:
        for i in range(0,3):
            t.fd(length); t.left(120) # draw triangle
    else:
        draw_sierpinski(length/2,depth-1)
        t.fd(length/2)
        draw_sierpinski(length/2,depth-1)
        t.bk(length/2)
        t.left(60)
        t.fd(length/2)
        t.right(60)
        draw_sierpinski(length/2,depth-1)
        t.left(60)
        t.bk(length/2)
        t.right(60)
window = turtle.Screen()
t = turtle.Turtle()
draw_sierpinski(100,2)
window.exitonclick()
```



References

K. J. Beers. *Numerical Methods for Chemical Engineering: Applications in MATLAB*. Cambridge University Press, New York, NY, 2006. ISBN 9780521859714.

G. Blanchet and M. Charbit. *Digital Signal and Image Processing using MATLAB®*. ISTE Ltd, 2006.

S. L. Campbell. *Modeling and Simulation in Scilab/Scicos*. Springer Science+Business Media, Inc., 2006.

I. Danaila, P. Joly, S. M. Kaber, and M. Postel. *An Introduction to Scientific Computing: Twelve Computational Projects Solved with MATLAB*. Springer Science+Business Media, LLC, 2007.

P. Deuflhard and A. Hohmann. *Numerical Analysis in Modern Scientific Computing: An Introduction*, volume 43 of *Text in Applied Mathematics*. Springer-Verlag New York, Inc., 2 edition, 2003.

S. Elnashaie and F. Uhlig. *Numerical Techniques for Chemical and Biological Engineers Using MATLAB®: A Simple Bifurcation Approach*. Springer Science+Business Media, LLC, 2007.

F. R. Giordano, W. P. Fox, S. B. Horton, and M. D. Weir. *A First Course in Mathematical Modeling*. Brooks/Cole Cengage Learning, Singapore, 4th edition, 2009.

N. J. Giordano and H. Nakanishi. *Computational Physics*. Pearson/Prentice Hall, Upper Saddle River, NJ, 2nd edition, 2006.

J. Grus. *Data Science from Scratch: First Principles with Python*. O'Reilly Media, 2015. (A nice book with many examples).

V. Haenel, E. Gouillart, and G. Varoquaux. Python scientific lecture notes release 2013.2 beta (euroscipy 2013). `https://www.scipy-lectures.org/_downloads/PythonScientific-simple.pdf`, September 2015.

M. T. Heath. *Scientific Computing: An Introductory Survey.* McGraw-Hill, 2nd ed. edition, 1997.

Y. Hilpisch. *Python for Finance: Analyze Big Financial Data.* O'Reilly Media, Inc., 2015.

E. Holzbecher. *Environmental Modeling Using MATLAB®.* Springer-Verlag Berlin Heidelberg, 2007.

J. Kiusalaas. *Numerical Methods in Engineering with Python.* Cambridge University Press, 2005.

H. P. Langtangen. *Python Scripting for Computational Science.* Springer Verlag, 2004.

H. P. Langtangen. *A Primer on Scientific Programming with Python*, volume 6 of *Texts in Computational Science and Engineering.* Springer-Verlag Berlin Heidelberg, 2009.

D. C. Montgomery and G. C. Runger. *Applied Statistics and Probability for Engineers.* John Wiley and Sons, Inc., 3rd edition, 2003.

A. Quarteroni and F. Saleri. *Scientific Computing with MATLAB and Octave.* Springer-Verlag Berlin Heidelberg, 2nd edition, 2006.

W. Richert and L. P. Coelho. *Building Machine Learning Systems with Python.* Packt Publishing, 2013.

M. Swamynathan. *Mastering Machine Learning with Python in Six Steps: A Practical Implementation Guide to Predictive Data Analytics Using Python.* Apress, 2017.

A. Tveito, H. P. Langtangen, B. F. Nielsen, and X. Cai. *Elements of Scientific Computing.* Springer-Verlag Berlin Heidelberg, 2010.

S. Vaingast. *Beginning Python Visualization: Crafting Visual Transformation Scripts.* Apress, 2009.

G. Varoquaux, E. Gouillart, O. Vahtras, C. Burns, A. Chauve, R. Cimrman, C. Combelles, P. de Buyl, R. Gommers, A. Espaze, et al. Scipy lecture notes. `https://www.scipy-lectures.org/_downloads/ScipyLectures-simple.pdf`, 2017.

M. M. Woolfson and G. J. Pert. *An Introduction to Computer Simulation.* Oxford University Press, 1999.