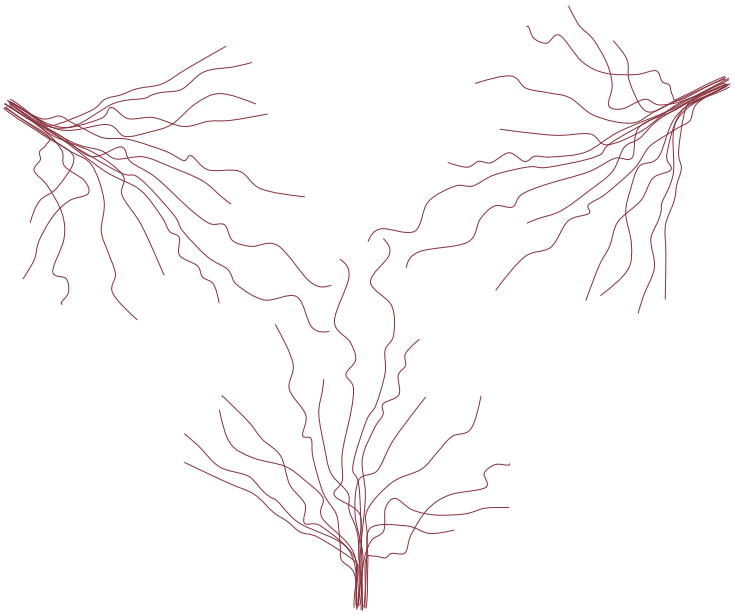
Second Draft

Networking for Software Developers

Lab manual to accompany COMP216.



Narendra K. Pershad

Software Group, ICET Dept.

SETAS >

Centennial College

Table of Contents

Introduction

Networking for Software Developer or COMP216 replaces CNET124. In contrast to CNET124, it emphasizes the software aspects more than the hardware aspects of networking. You will learn how to write code to connect to and send HTTP command to a Web Server and how to send mail to and retrieve mail from mail servers. You will understand what a socket is and how to use a socket to perform some really cool things. Finally, you will build a real world IoT Solution.

This course is offered in the second semester of the fast-track and fourth or sixth semester of the normal non-technician program. It is expected that students have already settled down to the routines of college life, of managing studies and assignments and extra-college tasks. So, for this and all the course at Centennial College a reasonable amount of dedication is required from the student in order to be successful in this course.

Everything will be done using the Python programming language. You will learn to use one to the fastest growing language in the scientific and business community. A language that sports hundreds of thousands of packages/libraries/frameworks that are used in image processing, data analytics, machine learning and much more. These libraries/frameworks are often created by open-sourced minded individuals whom are experts in their fields. Hopefully, it will be a language that will be an asset in your future as a software developer.

At the end of the course it is hoped that you will have an understanding of the protocols that are used on the internet and in the interaction between two computers. You will also have an appreciation of the applications that operates on the internet and vulnerabilities of each protocols that they used and what can be done to mitigate them.

The delivery of this course is not coupled to any learning platform. No delivery method can and would be able to compare to in person delivery in an actual lab. However, the lab exercises were designed so that any student with a reasonable computer and with an internet connection will be able to complete all the assigned works to successfully clear this course.

Python as well as all the software that you will be using are open-sourced. You do not need to purchase any of the libraries that is needed for this course. What is not available as open-source software you will write/build yourself.

Operating System

Labs were developed using MacOS Catalina and tested on Windows 10. So, it is expected that both of these operating systems are fine to work with. I would venture that even any normal recent Linux distros would also be okay.

Python version

Labs were developed using version 3.6.5 64-bit and test with version 3.8. Please try to constrain yourself to these versions. Earlier versions are not guaranteed to work flawlessly. All of the python libraries were installed using the python package manager pip.

Python virtual environment was not used because we feel that the benefits that you will get will not justify the added complexities involved. We will also need only a few packages that we will be used quite substantially. If you are comfortable with a virtual environment manager, you can continue working with it since there are benefits of developing in a controlled environment.

The completion of each lab will require very specific Python packages. A list of all the packages for all the labs is given at the end of this document.

IDE

Integrated **D**evelopment **E**nvironments' have evolved a far way. Initially, developers started with a simple text editor that was used to enter code in a file, then another program to compile the code, then another to link the output of the previous step to create an executable and finally run the executable. If there is an error at any stage, then you stop and make the corrections in the editor and then start the process all over again. An IDE does all of the above steps in one convenient interface and more. IDEs sports sophisticated systems that does code hinting, syntax checking with code profiling and single step execution and much more. It raises the productivity of the developer enormously as well as the quality of the resulting software.

It is advised that you use Visual Studio Code for all of your development. It is built from open-source software and it is multi-platform so there is a version that will run on your machine. It also supports all of the popular languages. It is light, fast and has a rapidly growing community of users. It is continually being updated and is supported with a plethora of extensions.

VS Code offers syntax coloring, code hinting, refactoring debugging and even execution of code within the IDE. It sports a modern key-bindings that will be more comfortable to windows users. There is a plugging that does linting (highlights code irregularities).

If you have a personal editor that you prefer to use such as *sublime text™*, *atom™*, *idle™*, *spyder™* or *pycharm™* then please understand I might not be able to support you as well as those using Visual Studio Code.

About this manual

We developed this manual from our experiences and our collective predictions on the current state and trends in the software world. It is written as a series of five modules that is meant to be completed in order because each module builds/relies on the concepts and techniques that were covered in the previous part(s). The code samples as well and the lab exercises gets progressively more complicated as the instructions get less verbose. It is hoped that this gradation will gently ween you onto the final module which requires polished application.

This course is in the latter part of your model route at Centennial, we don't assume any prior knowledge of python. However, we do expect that you have cleared Programming I and II, Unix Operation Systems and Software Engineering Methodologies. We hope that you understand the concept of control structures, classes, gui, sound defensive coding practices and design patterns.

We hope that you look pass the missteps of this work in your journey and make useful criticisms so that the next iteration of this work will be more complete and helpful to the next cohort of students using it.

What is not covered?

Although the content of this course was decided upon with great deliberation, IT field is very dynamic with new technologies been developed and old technologies evolving continuously so there will be debates on what should topics and should not be part of this course. As with all of the courses in our curriculum we are always looking as ways of keeping our course current and relevant.

The following concepts were in strong contention of being part of this course: software defined networks, networking automation, Bluetooth, near field communication, ldap, samba and authoring web api's and rest services. Maybe some time in the future, some of the above or something totally new will be incorporated in COMP216.

In terms of python language workflows, we will not cover the entire process of building a standalone application nor its installation nor testing.

How to use this manual effectively

This manual is the ultimate source for all the materials for COMP216, it will also serve as a guide as you progress in each week and each module. Each week starts with an overview which will comprise of:

- Weekly Learning Outcomes:
  What (list of) will be achieved at the end of this week.
- Checklist:
  What tasks needs to be done (readings, videos, PowerPoint slides, exercises and assignments) in order to satisfy the above WLO.
- Reading list:
  What is the required reading (recommended as well as supplemental) for this week.
- Due:
  What is exercise/labs/assignment is due at the end of the current week.

At the end of each week there is:

- Summary Checklist:
  A summary of this week's activities.
- What will covered in the next week:
  A brief look of what will be next week's topics.
- Reminder:
  Reminder of what should have been completed this week.

Expectations

It is expected that you complete all the assigned reading and labs by yourself. You may discuss your work with your colleagues or seek guidance from your instructor, but never ever submit someone's work as your own. That is plagiarism, and in this course as well as at Centennial College it as a serious offence which may result in dire consequences.

Almost all labs will have marks attached. There will penalties for missing or for late submissions. These will be discussed in your first lecture as well as the method of

submission for each work. As well, there will be a naming convention for your uploaded files.

It is expected that as you progress in the course, you become more in tuned with the intensions of the instructor and what is expected of you. The instructions will be less verbose and the tasks will be more challenging that will require more research and planning.

The use of search engines and programming forums is encouraged in this course. You are free to incorporate code from external sources in your submission, however, any code that you do not write yourself must be properly credited i.e. you must cite code that you use. It is expected that any submission must not contain more than 25% of external code. So any submission to your professor must have at least 75% of code written by you.

It is my fervent hope that during your journey, you will learn something that will be beneficial to your future and that the learning experience be enjoyable.

Happy networking and good luck.

Module 1

In this module you will survey the python language, you will learn about data types and control structure how to write a variety of programs. You will be writing short scripts as well as more complex programs. Applications that might sport a command line interface or a graphical user interface. You will look at the various ways to run your code and to troubleshoot and debug your programs.

You will look at OOP features such as classes and inheritance.

You will end this module by writing threaded applications and looking at advanced language features such as closures and higher order functions.

Week 1 – Python Basics

Learning Outcomes

- What is different about programming in Python.
- How to work with the language.
- The primitive data types available.
- Syntax of the various control structures.

Checklist

- Read this chapter.
- Work through the notebook examples
- Download and install Python and Visual Studio Code

Due this week

- .

Reading List

- The materials in this chapter.

The first three week of this course is not about teaching you programming, rather it is about you learning the python language, using python idioms and getting to know the rich ecosystem around the language to use the language efficiently.

This week we will look at the number of ways that you can execute python code, the built-in data types supported by the language and the control structures that allows you to make predictions about a block of code.

1.a – Basics

This part lists and explains the various ways that you may run your python code.

1.a.1 – Running your code

There are basically four ways of running python code: from a jupyter notebook, from the python repl, invoking the python executable and passing the code file and finally setting the permission mode of the code file. Each method has it convenience and use.

1.a.1.1 – Python REPL

This way is simple and it allows you to build your application incrementally while getting feedbacks promptly. It is very popular with beginners and even with veteran *Pythonistas*.

You will need to start the python interpreter, this exposes a command prompt similar to the one offered by most operation system shells. This is a Read Evaluate and Print Loop. You write code blocks that are evaluated and the results sent back to the console.

You can this technique to incrementally build substantial python applications or parts of applications. You are able to examine each object at your leisure. You may also debug problematic code this way.

The downside is that it is not convenient for larger/longer block of code or distributing code to end users or when typing is challenging.

```
$ python3
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 05:52:31)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 4 + 20
>>> print(f'The value of a is {a}')
The value of a is 24
>>>
```

1.a.1.2 – Running Python code from the command prompt

You will use this only after completing your entire application.

```
«path_to_python_interpreter» «name_of_code_file»
```

This should only be attempted when you are satisfied that your code is preforming according to specifications.

1.a.1.3 – Python 'executable'

In this method of running python code there are two steps involved, changing the permission of the file and then inserting the path to the python interpreter in the first line of the code file.

```
chmode 755 «name_of_code_file»
```

And then modifying the code file, insert the following at the top of the file:

```
#!«path_name_of_python_interpreter»
```

This is probable the best and the most professional way of running or distributing your code to everyone from the novice user to the seasoned developer.

1.a.1.4 – Jupyter Notebooks

This is rapidly becoming a popular way of running and demonstrating code to a wide category of users. The notebook consists of a set of three different kinds of cells: cell contain text in markdown format, cells with python code and cells with output from code. It is like browsing a live textbook, you can read the theory and see the code and finally view the results of executing the code without leaving the notebook. You may also modify the code and re-run to see the effects of the changes.

Online providers even have embedded environments that can follow the notebooks so the required libraries are available automatically. This is beneficial especially to novice user or when the environment is challenging to configure/setup more so when trying to learn new workflows such as trying machine learning or image analysis.

To save time in typing and trouble-shooting your code, the next part of your work has been composed in jupyter notebooks. If you have a google account, you may use log into your account and then proceed to the url `https://colab.research.google.com/` to complete these exercises otherwise you may use Visual Studio Code to work with them.

These notebooks may be accessed via:

https://colab.research.google.com/drive/1_dn47ox7Fowho7qKCcxFOHoPDkiFAhJJ?usp=sharing

https://colab.research.google.com/drive/1UK7IcfGjwP7RZIbkC8WE-8nIhhVC_zwo?usp=sharing

https://colab.research.google.com/drive/1H1WqfDHqdSB8hAHZo23fAyrJSS19qEK7?usp=sharing

Cloud support for jupyter notebooks includes the following entities: Binder, Kaggle, Google, Microsoft, CoCalc and Datalore.

1.b – Data Types

Before learning a new language, the first place to start is examining the primitive/building data-type supported by the language. Python, unlike the other structured languages you have studied so far is a dynamically-typed language (the other dynamically-typed language you are likely to encounter is javasript™). So a variable may contain different data types in the life time of a program.

Python supports the following data types:

1.b.1 – None:

This is similar to the **`Null`** object in C#. **None** is un-equal to everything except to another **None**.

1.b.2 – Numerics:

Numbers are immutable, this means that values in memory are not mutated, another spot is chosen with the new value.

There are three types of numbers available without any imports:

1.b.2.1 – int

Int have unlimited precision.

int literals

```
7                                   #base 10 value
0b100                               #base 2 value
0o27                                #base 8 value
0xf                                 #base 16 value

0b100_000                           #base 2 value
```

1.b.2.2 – bool

This is a numeric sub-type, having only two literal bool values:

`True` or **`False`**

Unlike C#, however any python object can be tested for truth value. The following values will be considered False:

- None,
- `False`
- Zero numeric type: `0`, `0l`, `0.0`, `0j`
- Any empty sequence: `''`, `{}`, `[]`

All other values are considered **`True`**

1.b.2.3 – float

Float literals

```
3.14, 10., .001, 1e100, 3.14e-10
```

1.b.2.4 – complex

Complex literals

```
3.14j, 10.j, 10j, .001j, 1e100j, 3.14e-10j
```

1.b.3 – Sequences:

The six sequences in python shares some common operations such as iterable and indexable:

- Strings (Binary and Unicode)
- Tuples -> ()
- Lists -> []
- stack, queue will not be covered in this course

Unlike most modern languages, python support negative indexing. If the index is positive, then it is interpreted (like other languages) position from the start of the sequence. If the index is negative, then position start from the end of the sequence. So an index of -1 will reference the last item in the sequence.

1.b.3.1 – Immutable

1.b.3.1.1 – Strings

Strings are text delimited by a matching pair of single, double or triple quotes. In this course, we will use a single quote for strings with a few exceptions. Triple quotes (either single or double) are used for multi-line string such as docstring.

1.b.3.1.2 – Tuples

A sequence of immutable python objects. This means that it (any of the items) may not be changed after assignment. The items do not have to be the same type.

```
>> a = ()                          #verbatim definition
>> a
()

>> b = (1)                         #this will not do a single item tuple
>> b
1

>> b = (1, )                       #now you have a single item tuple
>> b
(1,)

>> c = tuple('hello world'.split()) #tuple from list
>> c
('hello', 'world')

>> d = (1, 2) + (3, 4)             #concatenating two tuples
>> d
(1, 2, 3, 4)

>> e = (4, 5, 6) * 2               #repeats the original two times
>> e
(4, 5, 6, 4, 5, 6)

>> f = (['one', 2, [3], (4)])      #verbatim definition
>> f
('one', 2, [3], 4)
```

1.b.3.1.3 – Byte Strings

A byte string is a sequence of byte. This is not human-readable like a string which is a sequence of characters. This is the data type that is used in network communications

1.b.3.2 – Mutable

1.b.3.2.1 – Lists

A sequence of mutable python objects. This is the work horse in python for collection. It serves the purpose of arrays in other languages and it may contain any type of members. The following statements will return lists:

```
>> a = []                          #verbatim definition
>> a
[]

>> b = 'Hello world'.split()       #list from calling split on a string
>> b
['Hello', 'world']

>> c = list((3, 5))                #list from a tuple
>> c
[3, 5]

>> d = [1, 2] + [3, 4]             #concatenating two lists
>> d
[1, 2, 3, 4]

>> e = [4, 5, 6] * 2               #repeats the original two times
>> e
[4, 5, 6, 4, 5, 6]

>> f = ['one', 2, [3], (4)]        #verbatim definition
>> f
['one', 2, [3], (4)]
```

1.b.3.2.2 – Byte Arrays

Array of bytes

1.b.3.2.3 – Arrays

This is not used much in python.

1.b.4 – Set types:

Set (mutable) and Frozen sets (immutable).

A sequence of unique immutable python objects. Sets, list or dict cannot be in a set. Set does not support indexing.

Mimics a Mathematical set

1.b.5 – Mappings:

1.b.5.1 – Dictionary

A sequence of key-value pairs.

Each key must be unique and immutable such as strings, numbers, or tuples. A list may not be a key. There is no restriction on the value, it may be any python object. Dictionary does not support positional indexing, only key-indexing.

This data type is flexible and powerful and is extensively used in the language and will be heavily used in this course.

```
#there are lots of ways of creating a dictionary
>>> a = {'red': 'Danger', 1: 'first', 'second': 2, ('a', 'set'): 'a tuple is
my key'}


>>> b = dict(zip(['ilia', 'hao', 'yin'], ['Ilia Nika', 'Hao Lac', 'Yin Li']))


>>> c = dict([('arben', 'Arben Tapia'),('patrick','Patrick Gignac')])


>>> d = dict(nar='Narendra Pershad', joanne='Joanne Filotti')
```

Other data types includes methods, functions, classes, instances and exceptions. Methods are associated with objects and classes functions are not associated with a class.

1.b.6 – Data Conversion:

Most languages provides a mechanism to convert from one data type to another type either through implicit or explicit casting.

1.b.6.1 – Implicit

This kind of conversion is automatic e.g.:

```
a = 3.15                        #this is a float
b = 5                           #this is an int
c = a + b                       #b is up cast to a float before addition
```

```
#the following does not work (it does work in C#)
a = 'Hello'                     #this is a str
b = 5                           #this is an int
c = a + b                       #this does not work
```

1.b.6.2 – Explicit

Explicit casting is done by constructor methods such as `int()`, `float()` and `str()`:

`int()` will convert a float value or a value string to an int.

`float()` will convert an int value or a value string to a float.

`str()` will convert a float value or an int value to a string.

Other conversions can be done with: `tuple()`, `list()`, `set()`, `dict()`, `ord()`, `chr()`, `hex()`, `oct()`, `bin()`

1.c – Control Structures

Control structure dictates how the lines of code is processed: if it is sequential, if there are skips or jumps or if they are repetitions. It allows you to predict the result of a set of statements.

Each control structure brings some benefit(s) to the language.

1.c.1 – Sequence

This is the natural order of processing, you start with the first statement and then go to the next and continue until to get to the end. This is very simplistic but at the same time it serves as the foundations of most processing techniques.

Because all the statements are processed in the same order and there are no skips or jumps, this control structure is only able to complete one task. So additional control structure is needed to build any substantial program.

```
#set of statements without skips or repeats
>>> name = input('Please enter your name: ')
Please enter your name: Narendra
>>> program = input('Please enter your program: ')
Please enter your program: Game - Programming
>>> print(f'Hello {name} welcome to {program}')
Hello Narendra welcome to Game - Programming
```

The above code will always do the same thing: prompt for a string, reading the string prompt for a second string and read that also. Then it prints a message. Programs written with only sequence control structure is capable of completing only a single task.

1.c.2 – Conditionals

This control structure allows you to process or ignore code blocks depending (true/false blocks). This gives immense power to the language, now your programs are

able to complete multiple/different task. Simply implement the individual tasks in different blocks. If there are more task to be implement then nest more conditionals.

Unlike most other modern languages there is no switch in the python language.

Notice unlike C# there is no parenthesis around the assertion nor are there curly braces to delimit the true or the else code blocks. Blocks are defined by indentations. Most IDE's will manage indentations for you (VS Code use 4 spaces for an indent), but if you paste code from another source, then there could be problems. Visually you may not be able to discern between 4 spaces and a tab, the python interpreter will complain most vehemently.

1.c.2.1 – If statements

The *`if statement`* allows the optional processing of a code block. This implies that the same program is capable of completing two distinct tasks: one if the block is processed and another if the block is ignored.

```
#simple if statement without else block
if «assertion»:
    Body of if
```

```
#simple if statement with else
if «assertion»:
    Body of if
else:
    Body of else
```

```
#a single if statement
```

```
if «assertion»:
    Body of if
elif «assertion»:
    Body of elif
else:
    Body of else
```

```
#nested if statements
if «assertion»:
    Body of if
    if «assertion»:
        Body of inner if
    else:
        Body of inner else
else:
    Body of else
```

1.c.2.2 – Ternary statements

The *ternary statement* is a short hand if-else statement.

```
#ternary statement
«value if assertion is true» «assertion» «value if assertion is false»:

status  = 'Wealthy' if asset > 1_000_000 else 'Normal'
```

There is a short hand version

```
#ternary statement
>>> a
345
>>> a or 'Hello'
345
>>> a and 'Hello'
'Hello'
```

1.c.3 – Repetition

This provides the structure to process code blocks multiple times. Hence, you have to opportunity to write programs that might not have been logistically possible without repetitions. E.g. Printing the number from one to one million.

There are no *do-while loop* nor are there the normal *for loop*. Python supports the break and continue keywords.

1.c.3.1 – While statements

The while loop can be considered a conditional iteration. Do the loop whilst a condition is satisfied and stop when it is not.

```
while «assertion»:
...  Body of while
```

```
>>> x=1
>>> sum=0
>>> while x < 100:
...  sum += x
```

```
...  x += 1
...
>>> print(sum)
4950
```

1.c.3.2 – Using the range object

The range() function returns a range object which is an immutable sequence of numbers. A for loop is normally used to iterate this sequence. The range function takes one mandatory and two optional arguments.

```
range([«start=0»], «end», [«step=1»])
```

```
>>>for x in range(3, 10, 2):
...  print(x, end=' ')
...
3, 5, 7, 9 >>>
```

```
# if you are not using the value, then the linter
# will complain about un-used variables. The under
# score prevents this it is referred to a 'discard'.
>>>for _ in range( 10 ):
...  print('Hello', end=' ')
Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello
```

1.c.3.3 – Using an enumerable

Enumerable includes strings, lists, sets, tuples and dictionaries.

```
>>>a = [3, 5, 7, 8]
>>>for x in a:
...  print(x, end=' ')
...
3 5 7 8 >>>
```

```
>>> name = 'Narendra'
>>> for a in name:
...  print(a, end=' ')
...
N a r e n d r a >>>
```

1.c.3.4 – Using the Enumerate Function

The enumerate() function takes a collection (a sequence or mapping) and adds a counter and then return an enumerate object.

```
enumerate(«collection», [«start=0»])
```

```
>>> profs='ilia yin arben narendra mehrdad'.split()
>>> #process all the items
... for pos, name in enumerate(profs):
...   print(f'{pos} {name}')
...
0 ilia
```

```
1 yin
2 arben
3 narendra
4 mehrdad
```

```
# you can start the count at a particular integer
>>> profs='ilia yin arben narendra mehrdad'.split()
>>> for pos, name in enumerate(profs, start=2):
...   print(f'{pos} {name}')
...
2 ilia
3 yin
4 arben
5 narendra
6 mehrdad
```

```
# you can iterate a dictionary
>>> profs = {'ilia': 'COMP304', 'yin': 'COMP306', 'arben': 'COMP305',
'narendra': 'COMP216', 'mehrdad': 'COMP225'}
>>> for pos, name in enumerate(profs):
...   print(f'{pos}: {name}')
...
0: ilia
1: yin
2: arben
3: narendra
4: mehrdad
```

```
# a better way to iterate a dictionary is to use the .items() method of the
dictionary class
>>>
>>> for prof, course in profs.items():
...   print(f'{prof.rjust(12)}: {course}')
...
        ilia: COMP304
         yin: COMP306
       arben: COMP305
    narendra: COMP216
     mehrdad: COMP225
```

1.c.4 – Functions

This control structure gives the programmer the ability to attach a label to a block code. So, you can use that label whenever that logic is required. This facilitates the reuse of code, so less code is written which implies less chance of error.

You can consider a function as a mini program which should be tasked with a single responsibility. It may take inputs (called arguments) and produce outputs (return value(s)). Functions can call other functions or even itself (recursion).

There are many arguments for using functions, it forces you to decompose your problem into simpler tasks each with a single responsibility. So, the developer now has to implement a simple task with a single purpose which is easier to verify and test.

Python supports optional arguments (arguments with default values), keyword arguments (similar to named arguments) and arbitrary arguments (variable number of arguments). It also supports anonymous function definition which is definition functional units with attaching names.

Unlike most modern languages, python does not support function overloading i.e. multiple functions with the same name. If you code more than one function with the same name, then the later ones with over-write the earlier ones so there is only one active implementation. You can partially simulate function overloading by using default arguments.

1.c.4.a – Annotation

Python support function annotations for both parameter and return type. Completely optional, python does not attach any meaning or significance to it. However, it can be used by third-party applications to enhance their operations. IDE such as Visual Studio Code can provide type-checking and code hinting. It can be also be used for foreign-language bridges, predicate logic functions database query mappings etc.

```
def «function_name»(«list_of_argument(s)»):
    Body of function
```

```
>>> def hello(name: str) -> None:
...   print(f'Hello world, from {name}')
...
>>>
>>> hello('ilia')
Hello world, from ilia
```

The above example demonstrate “type hinting”, and it makes your code more user friendly. Although type hinting is optional we will try to use it as much as possible in this course.

1.c.4.1 – Inner Functions

A function that calls declared within another function is an inner function or a nested function. The inner function has access to the variable of the enclosing function so it is often used as helper functions. It can also be used to provide encapsulation or hiding the inner function. Unless the inner function is extremely short it is a better practice to use private helper functions. When deciding to use inner function, you should consider code readability and maintainability. You might think of using lambda expression instead.

Another popular use of inner function is closures that is covered in the part of this section.

```
def talk(phrase: list) -> None:
    def say(word: str) -> None:      #inner function
        print(word)

    words = phrase.split()
    for word in words:
        say(word)

>> talk('COMP216 is the best course')

COMP216
is
the
best
course
```

1.c.4.2 – Recursive Functions

A function that calls itself is a recursive function. A popular example of recursion is the factorial function:

$F_1 = 1$

$F_n = n * F_{n-1}$

```
def fact(n: int) -> int:      #annotations
    if n == 1:
        return 1
```

```
    else:
        return n * fact(n-1)

n = 5
print(f'The factorial of {n} is {fact(n)}')
```

$F_0 = 0$

$F_1 = 1$

$F_n = F_{n-2} + F_{n-1}$

```
def fib(n: int) -> int:
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
n = 8
print(f'The {n}th fibonacci number is {fib(n)}')
```

1.c.4.3 – Higher-order Functions

In Python functions are objects and like normal object, it can be stored as variables, passed to other functions or can be returned from functions.

```
'''
Created by Narendra for COMP216 (Sept 2020)
wk01c1_higher_order.py
```

```
Write a function that takes a function as an argument.
'''

def shout(text: str) -> str:      #annotations
    return text.upper()

def whisper(text: str) -> str:
    return text.lower()

txt = 'Hi Narendra'
shout(txt)                        #HI NARENDRA
yell = shout                      #assigns function to a variable
yell(txt)                         #HI NARENDRA

whisper(txt)                      #hi narendra

def greet(func) -> str:           #function that takes a function argument
    return func('from a higher power')

print(greet(shout))               #FROM A HIGHER POWER

print(greet(whisper))             #from a higher power
```

```
'''
Created by Narendra for COMP216 (Sept 2020)
wk01c2_higher_order.py

Write a function that returns a function.
'''

def make_entity(kind) -> (str):
    def client():                 #a nested function definition
        return 'I am a client'

    def server():
```

```
        return 'I am a server'

    if kind == 'client':
        return client
    else:
        return server

kind = 'client'
f = make_entity(kind)
print(f())                          #I am a client

kind = 'some_thing'
f = make_entity(kind)
print(f())                          #I am a server
```

1.c.4.4 – Closures

A closure is a function object (a function that behaves like an object) that remembers values in enclosing scopes (variables etc.) even if they are not present in memory. An inner function is able to access the (non-local) variable of the outer function.

A closure must satisfy the following:

- There should be nested function i.e. function inside a function.
- The inner function must refer to a non-local variable or the local variable of the outer function.
- The outer function must return the inner function.

```
'''
Created by Narendra for COMP216 (Sept 2020)
wk01c3_closure.py

Write a function that returns a closure.
'''
```

```
def adder(x) -> (int):
    def add(y) -> int:
        return x + y

    return add                    #returns a closure of the captured x

incrementBy2 = adder(2)           #a closure that adds 2
incrementBy3 = adder(3)           #a closure that adds 3
incrementBy9 = adder(9)           #a closure that adds 9

print(incrementBy2(3))            #5
print(incrementBy3(7))            #10
print(incrementBy9(30))           #39
```

1.c.4.5 – Built-in Functions

These are functions that are available in all instances of python without you explicitly coding it. They are very useful for both the novice and the expert programmer. Some of the more useful ones are:

- `id(x)`: returns the id of the argument. In cpython (the normal implementation) it is normally the object's memory address.
- `type(x)`: outputs the underlying type of the argument.
- `del(x)`: removes the argument from the current environment.
- `dir(x)`: outputs all the members (data as well as functional attributes)of the argument.
  If this function is invoked without any arguments, it will list all the variables in the current environment.
- `help(x)`: outputs same the same as the above with additional explanation on each member.
- `print(x)`: outputs the stringified version of the argument.
- `isinstance(obj, type)`: returns a bool indicating if the first argument is of type of the second argument.
- `hasattr(obj, attr)`: returns a bool indicating if the first argument has a member of type of the second argument. The second argument is of type string.

- `getattr(obj, attr)`: returns the value of the attribute specified by the second argument. If the attribute does not exist on the object then an exception is raised.
- `dir(__builtins__)`: returns a list of all the built-in function available in the default environment of the current python distribution.

Some other useful functions are: `str()`, `int()`, `float()`, `abs()`, `round()`, `max()`, `min()`, `len()`, `pow()`, `ord()`, `strip()`, `upper()`, and many, many more.

1.d – More on Collections

In this section we will summarize the four main collections

Set	Tuple	List	Dict
	`()`	`[]`	`{}`
`{'a'}`	`('a',)`	`['a']`	`{1: 'a'}`
`set('a')`	`tuple('a')`	`list('a')`	`dict({1: 'a'})`
`not indexable`	`[0]`	`[0]`	`[0]`
`.add('a')`	`immutable`	`.append('a')`	
`.update({'b'})`	`immutable`	`.extend(['b'])`	`.update(['b'])`
`.pop()`	`immutable`	`.pop()`	`.popitem()`
`.remove('a')`		`.remove('a')`	
	`.count('b')`	`.count('b')`	
`.clear()/ .discard()`	`immutable`	`.clear()`	`.clear()`

The following functions work on most of the above collections: `len()`, `max()`, `min()`, `sum()`,

1.e – Keywords

Every language has its own list of words that has special meaning or prescribe purpose. This list changes in python. To get a list of keywords for your current python distribution use `help('keywords')`.

The following is a list as of version 3.8

```
False       await       else        import      pass
None        break       except      in          raise
True        class       finally     is          return
and         continue    for         lambda      try
as          def         from        nonlocal    while
assert      del         global      not         with
async       elif        if          or          yield
```

Summary

Summary Checklist

This week we saw that python code can be executed in at least four ways:

- By using the REPL,
- By saving your code in a file and then using the interpreter to run the file,
- By creating an executable file and using bash to run the resulting file
- By using jupyter notebooks.

All four ways have advantages in different situations. The best way for a particular situation is the one that you are most productive at.

- We also looked at python built-in data types:
  - None
  - Numerics
  - Sequences
  - Sets
  - Mappings
- And control structures:

- Sequence.
- Conditional.
- Repetition.
- Functions

This first week completes a survey of the basic features of the Python language.

Next Week

Next week we will look at some more python language idioms and writing more capable programs. We will look at classes and objects and use exception handling to write more robust code. We will also use existing code as modules and finally how to write a command line (cli) program.

Reminder

- Lab 1 is due.

Week 2 – Python

Learning Outcomes

- How to write classes
- How to handle exceptions
- Writing a module. And the various ways of importing a module
- Create a command line interface application.

Checklist

- Read this chapter.
- Study the PowerPoint slides.

Due this week

- Lab 1 building a Class.

Reading List

- The materials in this chapter.
- PowerPoint

This week we will look at some other language features such as classes, exception handling, building and using modules and developing a **C**ommand **L**ine **I**nterface program.

2.a – Classes

Classes provide a way to bundle data (attributes) and functionality (methods) as a unit. You may declare class members as well as instance members.

2.a.1 – Creating a trivial class

The keyword class is used to create a class. Of course, all objects created from this class will have identical name values.

```
# creating a simple class

class Professor:
    name = 'Ilia Nika'
```

2.a.2 – Using a trivial class

We will instantiate the class and use the resulting object.

```
>>> ilia = Professor()           #notice the absence of the new keyword
>>> print(ilia.name)
Ilia Nika
```

```
>>> hao = Professor()          #instantiate the Professor class
>>> print(hao.name)
Ilia Nika

>>> hao.name = 'Hao Lac'      #change the name field of the object
>>> print(hao.name)
Hao Lac
```

2.a.3 – Creating a non-trivial class

The keyword class is used to create a class. This example illustrates lots of concepts and good coding practices.

1) The class and each of its members has an associated docstring.
2) Private members are denoted by a double underscore (e.g. `__id`)
3) Each instance method has a **`'self'`** argument. (C# also has the **`this`** but you don't need to supply it, it is implied.) Self has the same use as this is C#.
4) The **`__init__()`** is the constructor for the class. In C# and other popular languages, the constructor has the same name as the class.
5) Overloading the **`__str__(self)`** method enables you to decide how an object will be printed. (**`.Net`** looks for the ToString() method when an object has to be displayed.)
6) Overloading the **`__eg__(self, other)`** method allows you to dictate how the equality must be implemented. The default implementation is to compare the hash number of two objects and sometimes this is not the behaviour that you intend. So, you are free to implement what behavior is suitable to you.

```
# wk02a1_person.py


class Person:
    '''
    This is a python class.
    '''


    __id = 100000                    #class variable


    def __init__(self, name):        #constructor of any class is __init__
        '''
        This constructor initializes an object of this class.
        In C# the constructor must match the name of the class.
        '''
        self.name = name             #define a python property name
        self.id = Person.__id        #assigs a unique value to the attribute
        Person.__id += 1             #increments the class variable class
```

```
    def __str__(self):
        '''
        Return a string representation of this object.
        Similar to overriding the ToString method in C#
        '''
        return f'{self.id}: {self.name}'

    def __eq__(self, other):
        '''
        Overrides the == operator.
        '''
        return self.name == other.name
```

2.a.4 – Using the non-trivial class

We will instantiate the class and use the resulting object. Notice the absence of the **new** keyword.

```
>>> hao = Person('Hao Lac')  #You only need one argument for this constructor
>>> print(hao)
100000: Hao Lac
```

```
>>> ilia = Person('Ilia Inka')
>>> print(ilia)
100001: Ilia Inka
```

```
>>> prof = Person('Hao Lac')
>>> print(prof)
100002: Hao Lac
```

```
>>> alias = ilia
>>> print(alias)
100001: Ilia Inka
```

```
>>> print(f'{ilia} {"=" if ilia== alias else "!="} { alias }')
100001: Ilia Inka = 100001: Ilia Inka
```

```
>>> print(f'{ilia} {"=" if ilia==hao else "!="} {hao}')
100001: Ilia Inka != 100000: Hao Lac
```

```
>>> print(f'{hao} {"=" if hao==prof else "!="} {prof}')
100000: Hao Lac = 100002: Hao Lac
```

Try to run the following at the python prompt:

```
>>> type(Person)

>>> dir(Person)

>>> help(Person)
```

Where are the outputs coming from?

Are the output helpful/useful?

2.a.5 – Other concepts

This section explore some features that you would have covered when working with classes in C# programming, and are present in Python.

2.a.5.1 – Properties

As in csharp, a property is a hybrid of a method and a field. The user consumes it as a field and the developer publishes it as a method via a decorator. The following illustrates the definitions of both the getter and the setter of a property Age inside a class definition.

Currently it is not possible to define a setter without a corresponding getter.

```
@property
def Age(self):
  return self.__age
```

```
@Age.setter
def Age(self, age):
  self.__age = age if age > 0 else 1
```

2.a.5.2 – Class methods

Python distinguishes between a static and a class method. Static methods are methods in a class that does not use any member of the class, it is just there for convenience. Class methods get the implicit class as a reference to the method.

```
```

```
@staticmethod
def Add(a, b):
  return a + b
```

```
@classmethod
def BranchInfo(cls):
  return cls.__address
```

2.a.5.3 – Class attributes

Sometimes you an attribute to have the same value for all instances of a class. This can be achieved by a class attribute. The example below illustrates this.

```
class Course:
  DAYS = 'Mon Tue Wed Thu Fri Sat Sun'.split()

  def __init__(self, name, day):
    self.name = name
    if day in Course.DAYS:
      self.day = day
    else:
      raise Exception(f'ERROR: {day} is not a valid day')
```

2.a.5.4 – Overloading operators

Sometimes you might want to make your code easier to work with for example instead of having a method called add() or remove() you may choose to redefine the + and the - operator.

Some common methods that you may implement are:

Method	Notes
__str__	To achieve a sensible representation of an object
__eq__	Re-define the default behavior of the comparison operator
__add__	To implement addition
__subtract__	To implement subtraction

```
'''
Created by Narendra for COMP216 (Sept 2020)
wk02a1_class_person.py

Create a simple person class.
'''

class Person:
  '''
  This is a python class.
  '''

  __id = 100000                          #class variable

  def __init__(self, name):              #constructor of any class is __init__
    '''
    This constructor initializes an object of this class.
    In other languages, the constructor must match the name of the class.
    '''
    self.name = name                  #define a python property name
    self.id = Person.__id             #assigs a unique value to the attribute
    Person.__id += 1                  #increments the class variable class
```

```
  def __str__(self):
    '''
    Return a string representation of this object.
    Similar to overriding the ToString method in C#
    '''
    return f'{self.id}: {self.name}'

  def __eq__(self, other):
    '''
    Overrides the == operator.
    '''
    return self.name == other.name



#testing code
hao = Person('Hao Lac')
print(hao)
ilia = Person('Ilia Inka')
print(ilia)
prof = Person('Hao Lac')
print(prof)
friend = ilia
print(friend)

print(f'{ilia} {"=" if ilia==friend else "!="} {friend}')
print(f'{ilia} {"=" if ilia==hao else "!="} {hao}')
print(f'{hao} {"=" if hao==prof else "!="} {prof}')
```

The output of the above code is:

```
100000: Hao Lac
100001: Ilia Nika
100002: Hao Lac
100001: Ilia Nika
100001: Ilia Nika = 100001: Ilia Nika
100001: Ilia Nika != 100000: Hao Lac
```

```
100000: Hao Lac = 100002: Hao Lac
```

Finally some advice on operator overloading. Firstly you examine the behavior that you wish to implement and then you will define your method .e.g.

Behavior	Method to define
obj1 == obj2	def __eq__(self, other) -> bool: return self.some_property == other.some_property
obj = obj1 + obj2	def __add__(self, other): return «a new object based on the combined properties of obj1 and obj2»

obj = obj + value	def __add__(self, value):
	  mutate self according value
	  return self

2.a.6 – Adding attributes to and removing attributes from an object

Consider the object hao in the previous example. We will add an attribute **courses** and remove the attribute **id**. Other languages such as java, C++ and C# does not have an easy way to do this at runtime.

```
>>> dir(hao)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
```

```
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', '__weakref__', '_id', 'id', 'name']


>>> hao.courses = ['COMP231', 'COMP228']
>>> del(hao.id)
>>> dir(hao)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', '__weakref__', '_id', 'courses', 'name']
```

2.b – Inheritance

Inheritance and composition, allows you to build more complex/capable classes. Unlike most modern programming languages, python supports multiple inheritance i.e. the concept of a child class having more than one parent classes.

Inheritance enables you to define new classes by using the members of existing class(es). You have the option of either using the original method implement or defining a new implementation. You may also define additional data members.

2.c – Exceptions

When working with python you will get errors that falls into two categories:

2.c.1 – Syntax Errors

Syntax or parsing errors are probable the most common type of errors you get while learning python. These errors conflicts with the grammar definition of the language and can be detected by most IDEs. The parser prints the incorrect line with an arrow pointing to the earliest point where the error was detected. If you are executing the code via the python interpreter, then it prints the filename and line number also.

In compiled languages such as C++, C, java and C# most if not all syntax errors are caught before the executable is even built.

```
>>> if True print('Python is great')
  File "<stdin>", line 1
```

```
    if True print('Python is great')
                ^
SyntaxError: invalid syntax
>>>
```

2.c.2 – Exception

An exception is an event that disrupt the normal flow of the program instructions. Normally when the runtime encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents encapsulates everything about that error.

When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits. If the programmer anticipates an exception and does not handle it, then runtime terminates the program and prints what has happened.

```
#without handling
>>> a = 1 /0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```

```
#with exception handling
>>> try:
...   a = 1 /0
... except Exception as e:
...   print(f'Error {e} has occurred')
...
Error division by zero has occurred
>>>
```

Defensive programming is key to exception handling. If you anticipate that a code statement can result in an abnormal condition, then enclose that statement in a try block

2.c.2.1 – Exception Handling Syntax

The full syntax for exception handling is shown below:

```
try:
    pass                    #statements that may cause exception

except:
    pass                    #processed if that is an exception

else:                       #optional
    pass                    #processed if that is no exception

finally:                    #optional
    pass                    #guarantee to be processed even if there is no
exception
```

2.c.2.2 – Raising Exception

It is possible for a developer to deliberately create an exception.

```
#raising an exception
name = input('Please enter your name: ')
try:
    if name == 'Narendra':
        raise Exception('Narendra is evil!')
    else:
        print(f'Hello {name}')
except Exception as e:
    print(e)
```

2.c.2.3 – Custom-made Exception

It is possible to create your own Exception. Simply, create a class that inherits from the Exception class, just remember the invoke the base constructor in your constructor.

2.d – Modules and Packages

Module:

> A file that contains a collection of (related) functions and (supporting) global variables. It is simply a file with an extension of py file that has python executable code.
>
> We will be writing lots of modules in this course probable one for each lab in the later part of the semester. To access the logic in another file, simply insert a using statement at the appropriate position in your file.

Package:

> A collection of modules. It must contain an **`__init__.py`** file as a flag so that the python interpreter processes it as such. The __init__.py could be an empty file without causing issues.
>
> We will be writing a few packages probable for the last two assignments.

Library:

> A collection of packages.

Framework:

> A collection of libraries. This is the architecture of the program.

2.d.1 – Using modules

We will explore the various way of working with modules.

```
# wk02d1_utils.py
```

```
PI = 3.141592653589793


def add(x, y):
    return x + y

def diff(x, y):
    return (x - y) if x > y  else (y - x)

def area_of_circle(radius):
    return PI * radius * radius

def area_of_rectangle(width , length):
    return width * lengthimport numpy as np
```

The following show the interaction with the python REPL. All the functions in the module is imported under the name **`wk02d1_utils`**. Note the import statement does not require the file extension

```
$ python3
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 05:52:31)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__']
```

```
>>> import wk02d1_utils
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'wk02d1_utils']
```

```
>>> wk02d1_utils.add(3, 5)     #a very long way of invoking add()
8
```

The following import is easier to use, but it pollutes the environment.

```
>>> from wk02d1_utils import *
>>> dir()
['PI', '__annotations__', '__builtins__', '__doc__', '__loader__',
'__name__', '__package__', '__spec__', 'add', 'area_of_circle',
'area_of_rectangle', 'diff']
```

If you know which functions you are going to use, then the following import is probable the best to use.

```
>>> from wk02d1_utils import add, diff
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'add', 'diff']
```

The following are also possible. Try to figure out the results of the following imports:

```
>>> import wk02d1_utils as utils
```

```
>>> from wk02d1_utils import area_of_circle as area
```

```
>>> import myprogs.wk02d1_utils
```

2.e – cli Application

A command line interface application is built to just expose functionality. If the application sports only few features, or if it is a feature rich application, then the user has to be comfortable using the cli interface. Although it is difficult for a user to master these kinds of application, once the user becomes comfortable with the interface it is much faster to use and tend to like it better. Even for the developer, it is much simpler than creating a graphical user interface for your application.

The **sys** module offers limited ways of dealing with command line arguments. The **argparse** module is a provides a more sophisticated way of handling argument. There is built-in support for providing default arguments as well as error reporting and help. It is easier to design workflows and simpler to implement although requiring much more coding.

2.e.1 – argparse (simple example)

This is a trivial example.

```
# wk02c1_argument_parsing.py

import argparse

parser = argparse.ArgumentParser()    #creates the parser object
args = parser.parse_args()            #this can throw an exception
print(args)                           #prints all the arguments
```

Some interaction of the above code with the python REPL

```
$ python3 wk02c1_argument_parsing.py
Namespace()
```

```
$ python3 wk02c1_argument_parsing.py --help
usage: wk02c1_argument_parsing.py [-h]

optional arguments:
  -h, --help  show this help message and exit
```

```
$ python3 wk02c1_argument_parsing.py -narendra
usage: wk02c1_argument_parsing.py [-h]
wk02c1_argument_parsing.py: error: unrecognized arguments: -narendra
```

2.e.2 – argparse (a more complete example)

This is a more useful demonstration.

```
# wk02c2_argument_parsing.py

import argparse

DEFAULT_PORT = 12345                    #useful to have a default value
DEFAULT_PROTOCOL = 'tcp'                #useful to have a default value

parser = argparse.ArgumentParser()
parser.add_argument(                    #adds an argument
  dest='role',                          #mandatory argument
  help='either server or client')       #supplies help for this argument

parser.add_argument(
  '-proto',                             #optional argument
  choices=['http', 'tcp', 'udp'],       #restrict the choices
  default=DEFAULT_PROTOCOL,             #good to set default values
```

```
  help=f'can be http, tcp or udp')

parser.add_argument(
  '-port',
  default=DEFAULT_PORT,                 #sets the default value
  type=int,                             #specifies the type for this argument
  help=f'port to use in this communication. default {DEFAULT_PORT}')

parser.add_argument(
  '-user', '-u',                        #two ways to specify user
  help='the user name')

args = parser.parse_args()             #parses the argument
print(args)                            #prints all the arguments
```

Some interaction of the above code with the python interpreter

```
$ python3 wk02c2_argument_parsing.py
usage: wk02c2_argument_parsing.py [-h] -proto {http,tcp,udp} [-port PORT]
                                  [-user USER]
                                  role
wk02c2_argument_parsing.py: error: the following arguments are required:
role, -proto
```

```
$ python3 wk02c2_argument_parsing.py role=server
Namespace(port=12345, proto='http', role='server', user=None)
```

Because role is a mandatory argument, you may omit the name and supply the value as in the example below:

```
$ python3 wk02c2_argument_parsing.py server
Namespace(port=12345, proto='http', role='server', user=None)
```

```
$ python3 wk02c2_argument_parsing.py server -proto=tcp
Namespace(port=12345, proto='tcp', role='server', user=None)
```

```
$ python3 wk02c2_argument_parsing.py server -proto=mqtt
usage: wk02c2_argument_parsing.py [-h] [-proto {http,tcp,udp}] [-port PORT]
                                  [-user USER]
                                  role
wk02c2_argument_parsing.py: error: argument -proto: invalid choice: 'mqtt'
(choose from 'http', 'tcp', 'udp')
```

```
$ python3 wk02c2_argument_parsing.py --help
usage: wk02c2_argument_parsing.py [-h] [-proto {http,tcp,udp}] [-port PORT]
                                  [-user USER]
                                  role

positional arguments:
  role                  either server or client

optional arguments:
  -h, --help            show this help message and exit
  -proto {http,tcp,udp}
                        can be http, tcp or udp
  -port PORT            port to use in this communication. default 12345
  -user USER, -u USER   the user name
```

```
$ python3 wk02c2_argument_parsing.py server -port=1355
Namespace(port=1355, proto='tcp', role='server', user=None)
```

In this example, all the arguments are supplied:

```
$ python3 wk02c2_argument_parsing.py server -proto=tcp -port=4321 -
user=Narendra
Namespace(port=4321, proto='tcp', role='server', user='Narendra')
```

Summary

Summary Checklist

This week we covered the following:

- Classes: Defining classes with attributes and methods. Leverage inheritance to create new classes.
- Exception: Difference between syntax errors and Exceptions. Handling exceptions and raising your own exceptions.
- Modules: Writing a module and then including the logic in our environment via various forms of imports.
- Command Line Interface Applications: Writing an (non-graphical) application that is able to process command line arguments. We import a module (argparse) for the first time to simplify the processing of command arguments.

Next Week

Next week we will conclude surveying the python language by looking at some advanced features and using some built-in and external packages. We will take a brief look at the external `numpy` and `matplotlib` libraries and well as the internal `tkinter` and `threading` modules.

Reminder

- Lab 1 is due.

Week 3 – Python Advanced

Learning Outcomes

- Explore the numpy and matplotlib modules
- Create a Graphical User Interface application
- Build a multi-threaded application.

Checklist

- Read this chapter.
- Use pip to install the numpy and matplotlib modules
- Study the PowerPoint slides.

Due this week

- Lab 1 building a Python class.

Reading List

- The materials in this chapter.
- PowerPoint

This week we will end our python language excursion by looking at a few libraries, gui applications and multithreaded applications.

You will be using pip the python installer package to install the `numpy` and `matplotlib` libraries.

3.a – Numpy Library

Numpy is a python library that support large multi-dimensional arrays and matrices and a large collection of high-level mathematical functions to operate on them. It is invaluable for data science and machine learning and in many common workflows.

3.a.1 – Creating numpy arrays and converting to other types

```
# wk02a_numpy.py

import numpy as np

a = np.arange(
    3,                          #start (default is 0)
    10,                         #stop (mandatory)
    2)                          #step (default is 1)
a
array([3, 5, 7, 9])



b = np.linspace (
    1,                          #start (mandatory)
    2,                          #stop (mandatory)
    5)                          #number of gaps (default is 50)
b
array([1.  , 1.25, 1.5 , 1.75, 2.  ])

#creating a numpy array from a list
c = np.array ([1, 2, 3])        #takes a list
c
array([1 , 2, 3])



#creating a numpy array from a nested list
d = np.array([[1, 2], [3, 4], [5, 6]])
d
```

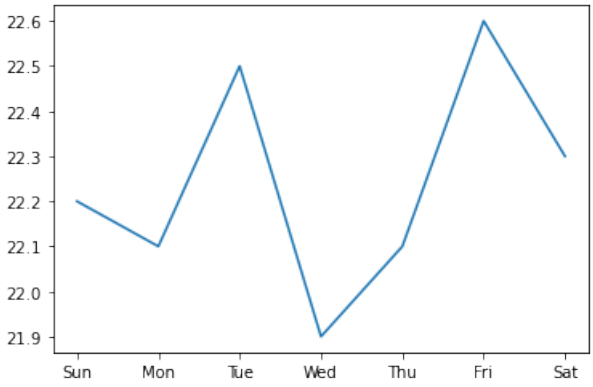
```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

3.b – Matplotlib Library

Matplotlib is a plotting library and is great complement to numpy. It can produce high quality graphics of a huge range of 2d and 3d diagram. It can even be animated.

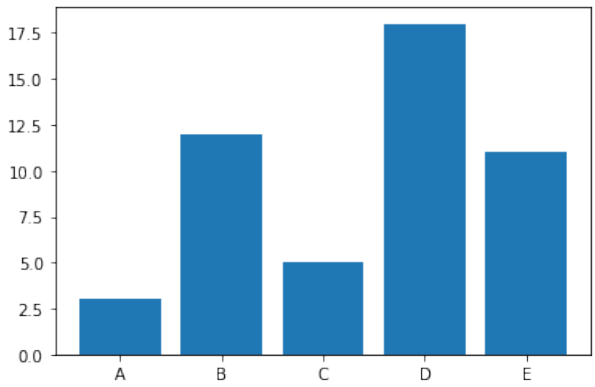
3.b.1 – Line Chart, Bar Chart, Pie Chart

```
#Line Chart

import matplotlib.pyplot as plt
temp = [22.2, 22.1, 22.5, 21.9, 22.1, 22.6, 22.3]
days = 'Sun Mon Tue Wed Thu Fri Sat'.split()
plt.plot(days, temp)
plt.show()
```

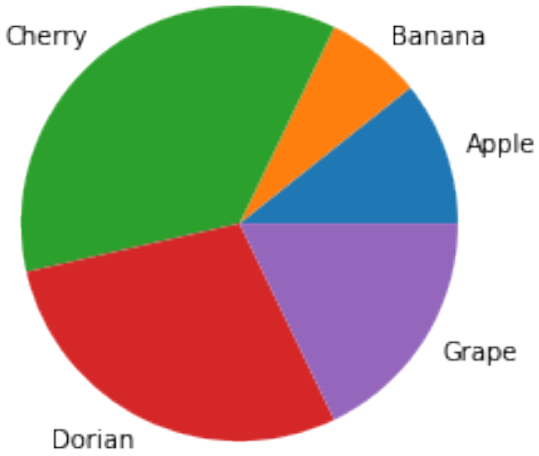



```
#Bar Chart

import matplotlib.pyplot as plt
height = [3, 12, 5, 18, 11]
bars = ('A', 'B', 'C', 'D', 'E')
plt.bar(bars, height)
```

```
plt.show()
```



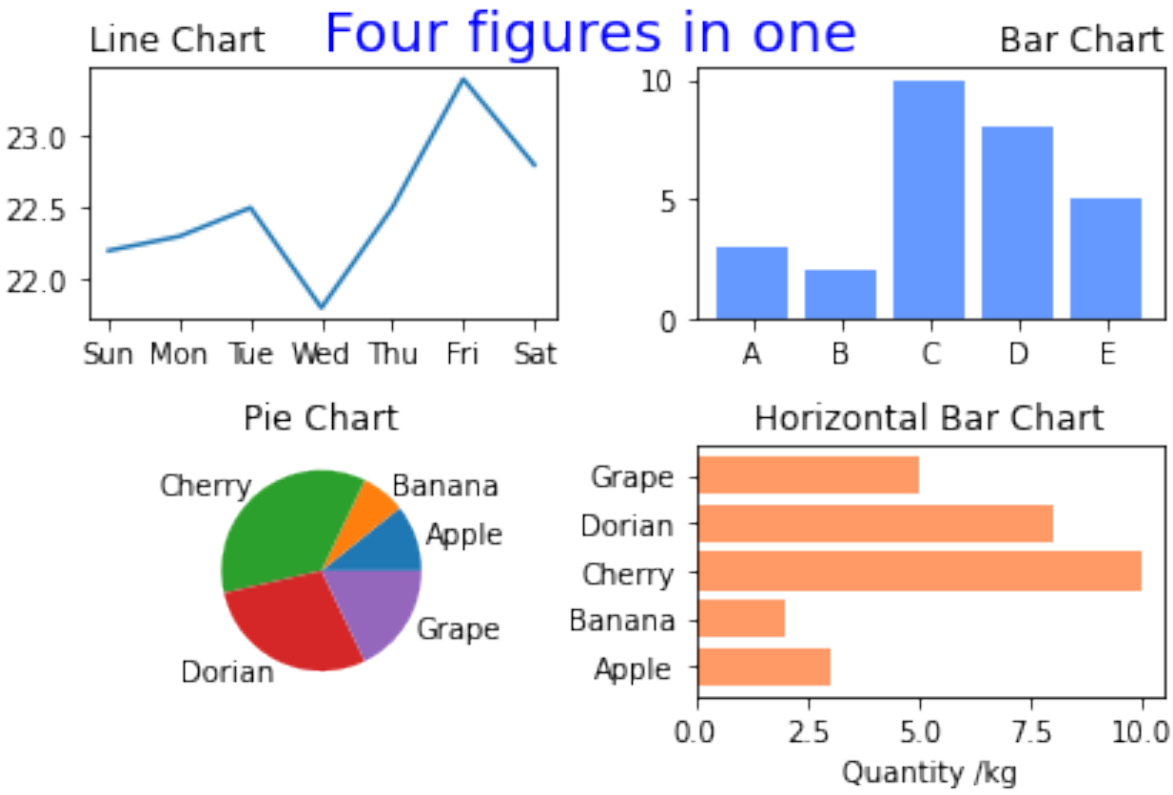

```
#Pie Chart

import matplotlib.pyplot as plt
weight = [3, 2, 10, 8, 5]
items = 'Apple Banana Cherry Dorian Grape'. split()
plt.pie(weight, labels=items)
plt.show()
```



3.b.2 – Sub-Plots

It is possible to display multiple plots on the same diagram. The following is a recipe for using sub-plots.

```
import matplotlib.pyplot as plt

plt.suptitle('Four figures in one', color='blue', fontsize=20)

# the first plot
plt.subplot(221)                                    #(rows, cols, pos)
plt.title('Line Chart', loc='left')
temp = [22.2, 22.3, 22.5, 21.8, 22.5, 23.4, 22.8]
days = 'Sun Mon Tue Wed Thu Fri Sat'.split()
plt.plot(days, temp)

# the second plot
plt.subplot(222)                                    #(rows, cols, pos)
…
…

# the third plot
plt.subplot(223)                                    #(rows, cols, pos)
…
…

# the fourth plot
plt.subplot(224)                                    #(rows, cols, pos)
…
…

# now show the result
plt.show()
```



3.b.3 – Animation

We will look at animation towards the end of the semester, when we will have to actually use it.

3.c – GUI Application

A GUI application as opposed to a cli application will always appear more polished and completed. The user feels more comfortable and more in control when using a gui. It is easier to design workflows and simpler to implement but it requires much more coding.

In this course we will be using **`tkinter`** package to build our gui. It is available in the standard installation of python. It originates from Tcl (Tool Command Language) and is a simple, fast, robust and platform independent windowing toolkit.

Although it lacks complex widgets such as rich text and HTML rendering, it is very stable and almost universally available wherever python is.

For this course all our application will be done via a class. We will create a class that inherits from Tk and define a constructor that will perform basic windowing task and at least one other method to create the user interface widgets. You may also create other methods as you see fit to make the application easier to implement and maintain.

3.c.1 – Window with a Label

```
# filename: wk03c1_gui_label.py
# by Narendra for COMP216 (Aug 2020)

# Creates a simple window with a
# single label

from tkinter import Tk
from tkinter.ttk import Label #themed Tk library

root = Tk()                   # like a csharp form
label = Label(                # csharp label
    root,                     #the parent of this widget
    text='Hello World')       #the caption of this widget
label.pack()                  #adds the label to the window

root.mainloop()               #start the main event loop
```

Running the above code will generate the window shown below:

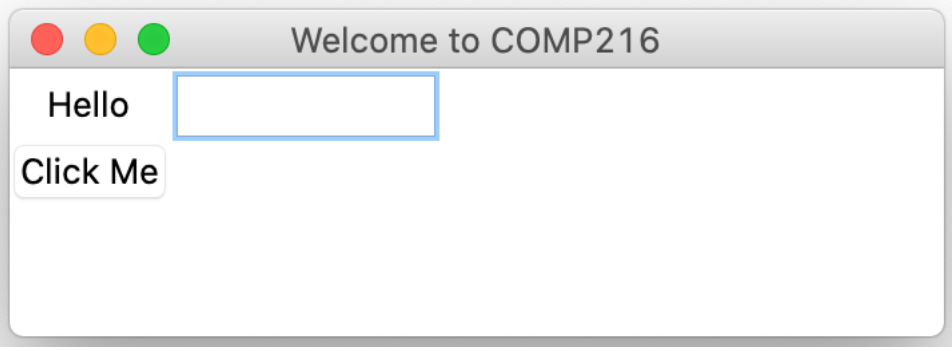
The above screen shot might look slightly different if you use another platform (I am using MacOS).

3.c.2 – Window with a Button, Label and Textbox

```
# filename: wk03c2_gui_button.py
# by Narendra for COMP216 (Aug 2020)

# Creates a simple window with a button
# label and a textbox. There is event
# handling
#
# https://likegeeks.com/python-gui-examples-tkinter-tutorial/
# the above url is a good intro to tkinter

# Button, label,  Entry (tkinter textbox), Combobox,
# Checkbutton, Radiobutton, ScrolledText, messagebox
# Spinbox, Progressbar, filedialog, Menu, Notebook (tab control)

from tkinter import Tk, Label, Entry, Button

root = Tk()
root.title('Welcome to COMP216')  #title of main window
root.geometry('350x100')          #the width and height of the window

lbl = Label(                      #
    root,                         # the parent of this widget
    text = 'Hello')               # the text on this widget
lbl.grid(
    column = 0,                   # the column position
    row = 0)                      # the row position

txt = Entry(
    root,                         # the parent of this widget
    width = 10)                   # the width
txt.grid(column = 1, row = 0)     # column and row position

def clicked():                    # this function will be wired-up to the
button
```

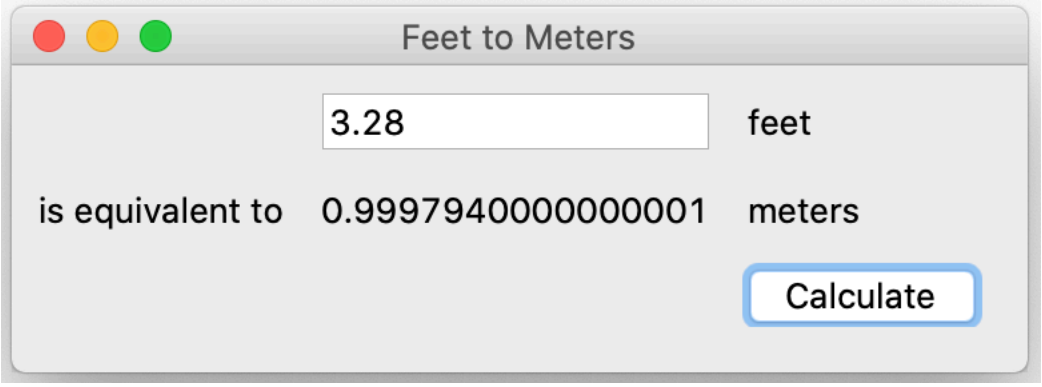
```
    res = 'Welcome to ' + txt.get()#get the text in the textbox
    lbl.configure(text = res)     # set the text on the label

btn = Button(
    root,                         # parent of this button
    text = 'Click Me',            # text on this button
    command = clicked)            # this is the event handler
btn.grid(column = 0, row = 1)

root.mainloop()
```

Running the above code will generate the window shown below:



3.c.3 – Window with grid layout

We will try to build the application shown below. There are six widgets arranged in an imaginary 3 by 3 grid layout. There is a single widget in the first column, two widgets in



the second column and three widgets in the third column. A better way of thinking and building the UI is to build it rows by rows starting at the top.

The code also shows the optimal way of working with string

```
# filename: wk03c3_gui_textbox.py
# by Narendra for COMP216 (Aug 2020)

# Uses labels, textbox and button.
# Event handler is attached to button and the window

from tkinter import Tk, W, E, N, S, StringVar
from tkinter import ttk


def calculate(*args):
    try:
        value = float(feet.get())
        meters.set((0.3048 * value * 10000.0 + 0.5)/10000.0)
    except ValueError:
        pass

root = Tk()
root.title('Feet to Meters')

# this will be the parent of subsequent widgets
mainframe = ttk.Frame(root, padding = '3 3 12 12')
mainframe.grid(column = 0, row = 0, sticky = (N, W, E, S))

root.columnconfigure(0, weight = 1)
root.rowconfigure(0, weight = 1)

# the following is the preferred way of working with strings
feet = StringVar()
meters = StringVar()
```

```
# row 0
feet_entry = ttk.Entry(mainframe, width = 7, textvariable = feet)
feet_entry.grid(column = 1, row = 0, sticky = (W, E))
#this is an alternate way of creating a label and setting grid coordinates
ttk.Label(mainframe, text = 'feet').grid(column = 2, row = 0, sticky = W)

# row 1
ttk.Label(mainframe, text = 'is equivalent to').grid(column = 0, row = 1,
sticky = E)
ttk.Label(mainframe, textvariable = meters).grid(column = 1, row = 1, sticky
= (W, E))
ttk.Label(mainframe, text = 'meters').grid(column = 2, row = 1, sticky = W)

# row 2
ttk.Button(mainframe, text = 'Calculate', command = calculate).grid(
    column = 2, row = 2, sticky = W)



for child in mainframe.winfo_children():
    child.grid_configure(padx = 5, pady = 5)

# set the focus to the textbox
feet_entry.focus()
# bind the enter key to the function calculate
root.bind('<Return>', calculate)

root.mainloop()
```

3.c.4 – Responsive Design

Responsive design refers to the automatic re-positioning and re-sizing of widgets when the host window is resized. This is the expected behavior of modern apps.

Newer apps have a responsive desi

3.c.5 – Some hints

1. Plan your layout on paper.
2. Use a grid-layout for all but the simplest of UI.
3. Place all the widget before implementing your logic.
4. Document your code excessively.
5. Use an OOP approach, although initially it is more difficult it will result in a much more focused implementation.

3.d – Multi-Threaded Application

Multi-tasking is the ability to do multiple jobs at the same time, this can be achieved by multithreading or multiprocessing.

Processes uses separate memory space, multiple CPU cores and bypasses GIL limitation in CPython. We will not write programs that uses multiple processes.

Threads are lightweight and they share the same memory space. The require less resources and are easier to share information and use. Applications that are I/O intensive such as reading or writing files, accessing a web-site, making request to a server are good candidates for multi-threading. Applications that are cpu intensive will NOT be good candidates for multi-threading.

We will use multi-threading in this course.

A thread can be defined as a sequence of instructions that can be run by a scheduler. Threads are lightweight processes (subparts of a large process) that can run concurrently in parallel to each other, where each thread can perform some task. Threads are usually contained in processes. More than one thread can exist within the same process. Within the same process, threads share memory and the state of the process.

The scheduler decides which thread will run and how long it will run. You don't have any guarantees of the order of the thread execution.

You may create a threaded application by:

a) Creating a thread directly and invoking the start() method on it
b) Sub-classing the `threading.Thread` class and implement the run method

3.d.1 – Calling a function synchronously

Firstly, we a function and then call it synchronously a few times. That means each call starts when the previous one ends.

```
# filename: wk03d1_threading_none.py
# by Narendra for COMP216 (Aug 2020)

# Runs a function five times synchronously

import time

def sleeper(n , name) :
    print(f'{name} going to sleep for {n} seconds')
    time.sleep(n)
    print(f'{name} done sleeping')


def do_five():
    start = time.perf_counter()
    for _ in range(5):
        sleeper(1, 'arben')

    end = time.perf_counter()
    print(f'Finished in {round(end-start, 2)} second(s)')

do_five()

print('program has terminated.')
```

Notice the program time is around 5 seconds

3.d.2 – Calling a function asynchronously (using threads)

Now we call the same function asynchronously multiple times. This means that all the calls starts at the same time and the os scheduler is responsible for running all of them.

```
# filename: wk03d2_threading.py
# by Narendra for COMP216 (Aug 2020)

# Creates a thread and invoke start

import threading, time

def sleeper(n , name) :
    print(f'{name} going to sleep for {n} seconds')
    time.sleep(n)
    print(f'{name} done sleeping')

start = time.perf_counter()
t1 = threading.Thread(            # creates a thread
    target = sleeper,             # function to run
    name = 'narendra',            # name for thread (optional)
    args = (1, 'narendra')        # function arguments as tuple
    )
t1.start()

profs = 'arben hao yin ilia vinay'.split()
thread_list = [t1]
for prof in profs:
    t = threading.Thread(
        target = sleeper,        # the name of the function to call
        name = profs,           # the name of the resulting thread (optional)
        args = (1, prof))        # the arguments to the function
                                 # no argument args = ( )
                                 # one argument args = (1, )
    t.start()
    thread_list.append(t)
print(f'{threading.active_count()} active thread(s)')
```

```
#this is to prevent further execution until all the thread completes
#for t in thread_list:
#    t.join()
end = time.perf_counter()

print(f'Finished in {round(end-start, 2)} second(s)')
print('program has terminated.')
```

Notice the program terminate almost instantly even though the function was called 6 times. To prevent the program from terminating before all the threads have completed un-comment lines (28 and 29.

The total execution time is about 1 seconds, this is because of the magic of threading, all the functions are executed at the same time in an asynchronous manner.

3.d.3 – Creating a threaded class

We will inherit from the `threading.Thread` class and override the run method. You will use this technique if a single function will not be able to capture the full logic of your application.

```
'''
Created by Narendra for COMP216 (Sept 2020)
wk03d3_threading_class.py

Create a threaded class.
'''

import threading, time

class Sleeper(threading.Thread):
    def __init__(self, n, name) -> None:
        threading.Thread.__init__(self)
        self.sleep_time = n
        self.name = name

    def run(self):
```

```
        print(f'{self.name} going to sleep for {self.sleep_time} seconds')
        time.sleep(self.sleep_time)
        print(f'{self.name} done sleeping')


start = time.perf_counter()

profs = 'narendra arben hao yin ilia vinay'.split()

#the following line uses list comprehension
thread_list = [ Sleeper(1, prof) for prof in profs]
for t in thread_list:
    t.start()                    #starting each thread

print(f'{threading.active_count()} active thread(s)')
for t in thread_list:
    t.join()                     #blocks until this thread ends
end = time.perf_counter()

print(f'Finished in {round(end-start, 2)} second(s)')
print('program has terminated.')
```

3.e – Closing remarks about python programming

This is the final section on learning/using the python programming language. The rest of the semester you can considered yourself as a python developer. You are expected to write code is a uniform way (this will be dictated by your current boss is who is your instructor). Also, you are expected to write, troubleshoot and debug code and to learn a new package all by yourself.

3.e.1 – Choice of Editor

A novice on the other hand will have a fundamental set of expectations such as line numbering, intelligent code hinting, syntax highlighting and checking, brace matching, auto-indenting, document outlining, multi-view.

As you become more seasoned you might opt for code refactoring, inserting code-snippets, auto-save, support for the cloud and versioning control. Later you will be interested in code profiling, emulation/simulation of the executing code, single-step code debugger, support for large project and collaboration.

As you become more proficient in a language, you will develop workflow and pattern so you will make more precise demands on your code editor. Other features that will improve your programming impressions are themes, macro support configurable keyboard mappings named templates, support for regular expression, source-code browsability, multi-platform and multi-language support and foldable code blocks and extensibility.

3.e.2 – Coding best practices

We read code a fair bit more than we write code so code should be written to be read! To ensure that code can be read by anyone, you need to follow the set of coding standards set out by your organisation (in this case your instructor). Above all you should be consistent in your commenting and documentation (avoid the obvious), formatting and follow the recommended naming scheme for variables, methods, classes, functions, files, folders.

3.e.3 – Growing as a python developer

You have completed at least five programming courses in at least three major languages so it is expected that you have acquired significant programming skills. These skills include but not limited to the following: fair level of technical proficiency, progressive work attitude, strong communication and people skills. It is also imperative that you be a team player with a focus on the big-picture, and respect deadlines and not hesitate to go the extra mile and able to handle failure. You should be comfortable in your work environment mastery and plan and prioritize your assigned tasks.

Summary

Summary Checklist

This week we covered the following:

- We have surveyed the `numpy` and the `matplotlib` libraries.
- We have also looked at building a graphical application using `tkinter`

- And then created threading applications.

This week brings us to the end of the first module which was a survey of the Python language.

Next Week

Next week we start a new module that will look at some network devices, the network software stack and the use of Wireshark™ to capture and examine network traffic.

Reminder

- Lab 2 is due.