NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

College of Computing and Data Science (CCDS)

CA6000 Applied AI Programming

Associate Professor

Nicholas Vun

aschvun@ntu.edu.sg

Preliminary

Course Aims and Learning Outcomes

This course aims to introduce appropriate programming fundamentals to students who will study other AI related courses in the MCAAI programmes.

Upon completion of this course, you should be :

- able to apply relevant features of Python programming language for AI related applications
- familiar to develop program using the Jupyter Notebook / Jupyter Lab platform
- able to utilize libraries and packages commonly encountered in AI applications
  - NumPy and Pandas for handling of data for AI/ML
  - Matplotlib and Seaborn to visualize data graphically
  - SciKit-Learn, TensorFlow, and PyTorch to perform basic training of AI models
  - LLM for low code programming

Course Structure

Seminar style with hands-on exercises in class

Weekly lessons

- One pre-record video(s) to watch before weekly lessons class
- Tuesday: 630pm - 830pm
- Saturday: 900am – 1230pm

Duration: 6 weeks

Assessment: 2 quizzes (Week 3 and Week 6) and 1 assignment

Python Basics

Python Programming Language

An easy to learn programming language that is widely used in many domains and has gain a lot of popularity in the recent years

- simple (English like) syntax
- vast collection of different libraries for AI/ML
- platform independence and broad community support
- many of the developments of DS and AI applications are now done in Python

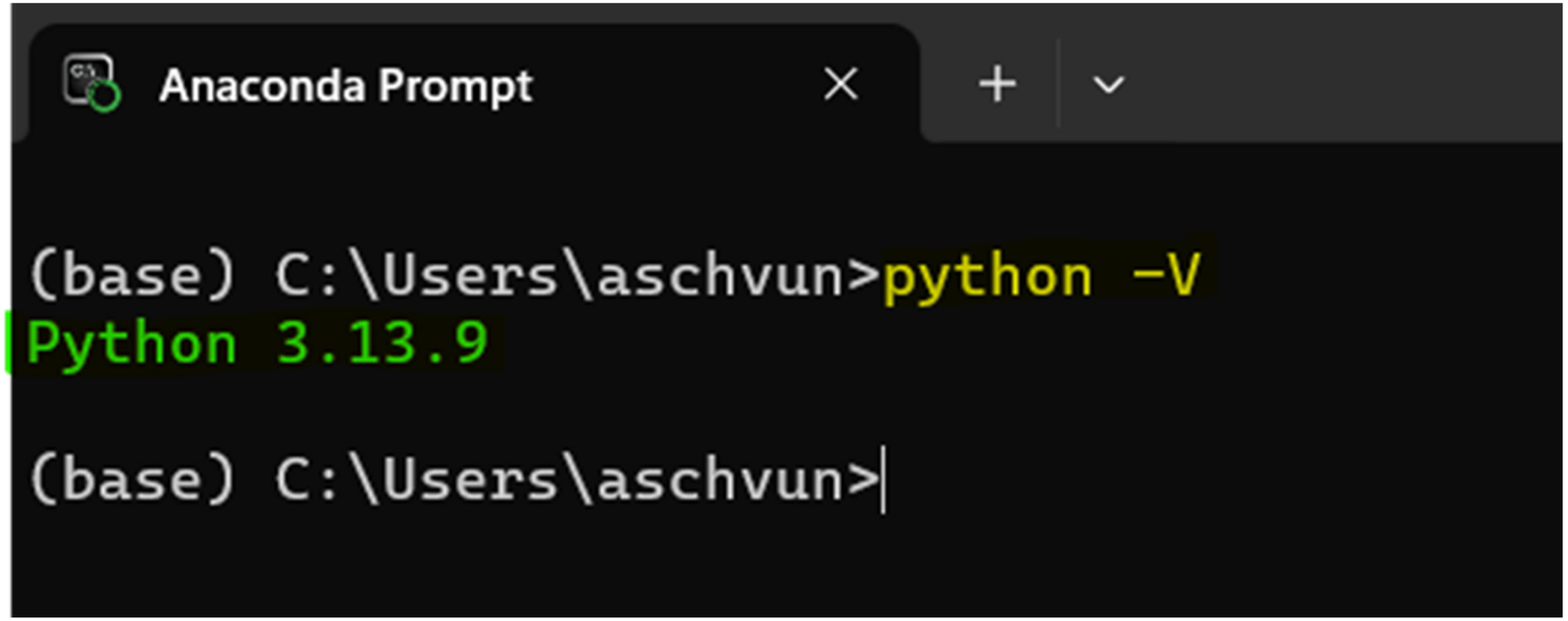
There are two versions of Python that are not exactly compatible

- Version 2
- Version 3.x ← you should use this

Python Interpreter

To start with, you will need to have a Python interpreter software to run python programs.

Check whether your computer is already installed with Python package



Basic User Input/Output Interface

User I/O Interaction - Output

Computer will only be useful if it can interact with the user

The simplest way to output messages to the user

- using the **print** function*

E.g., Key in the following Python code and press Enter key

```
>>> print ("Hello world")
```

This will display the corresponding message on the monitor.

- You would also have noticed that the code is executed (by the interpreter) as soon as Enter is pressed.

* more about 'function' later

User I/O Interaction - Input

A program will not be useful if users cannot interact with the program and enter their inputs

For input, Python 3.6 (and later) provides the `input()` function for this

- and use the `print()` function for text based out

```
name = input("What is your name? ")
print("Hello " + name)
```

The `name` used in here is known as a variable

Variable, Constant and Comment

Variables

Variables are ‘containers’ for storing data values

- allow the data to be retained and changed as needed
  - provides flexibility to manipulate it in our programme

Python has no command to declare (i.e. create) a variable

- a variable is automatically created the moment a value is first assigned to it using a statement

```
a = “Hello World!”
print (a)
b = 2 + 3
print (b)
```

The labels, a and b used is known as identifiers or names (of the variable)

Python Variables (cont.)

In Python

- variables do not need to be declared with any particular data type
- type can even be changed after they have been set

Examples:

```
a = 7
print (a)
a = “hello again”
print (a)
```

Variable Names

It is often useful to use an appropriate descriptive name for variable (in lower case)

- such that the programme can be more clearly understood by reader and ourselves

```
greeting = "Hello"
my_name = "Nick"
print(greeting, my_name)
```

Rules:

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Variable names are case-sensitive

Examples:

```
myName = "Nick"      #this is known as Camel case
my_name = "Nick"     #this is known as Snake case
```

Constants

Constants are variables whose values are intended to remain unchanged throughout the program's execution.

```
PI = 3.14159
MAX_MARK = 100
FILE_NAME = “test1.csv”
```

By convention

- constants are named using **uppercase letters (with underscores)** to signal their immutability.

Comment

Python has commenting capability for the purpose of in-code documentation.

- comment starts with a # symbol, and Python will render the rest of the line as a comment

```
#Test.py
greet_1 = "Hello"
greet_2 = 'World'    #single quote is also OK for string
```

Comments can be used

- to explain code and make the code more readable
  - to yourself in future and other that need to understand your code
- to skip execution when testing/debugging program code

```
#greet_1 = "Hello"
```

Docstring

For multiple line comments, enclose them using triple quotes (docstring):

```
''' or """
  '''
  Line 1 comment
  Line 2 comment
  '''
```

docstrings provide more comprehensive information then using comment

- about the purpose and usage of the code

Backslash \

The backslash \ is a special character in Python

When use with characters such as **\n**

- it form a new line character

```
print ("Hello, \n World!")
```

To insert special characters that may be 'illegal' in a string

- use the backslash escape character '\'

```
txt = 'It\'s alright.\n'
print(txt)
```

Backslash can also be used to end the expression on a line and continue it on the next line

- useful when having a (very) long expression that may not fit neatly in one line:

```
print ("Hello \
    World! \n")
```

Python
Data Types

Basic Data Types

Variables can store data of different types.

Some of the commonly used default built-in categories:

Str	String - a sequence of text/character
Int, Float	Numbers
Bool	Boolean – True/False
List, Tuple, Set, Range	Sequence (of items)
Dict	Dictionary

Examples of Data Types

`x = "apple"`	`string`
`x = 3.14`	`float`
`x = True or False`	`bool`
`x = ["apple", "banana", "cherry"]`	`list`
`x = ("apple", "banana", "cherry")`	`tuple`
`x = {"apple", "banana", "cherry"}`	`set`
`x = range(6)`	`range`
`x = {"name" : "John", "age" : 36}`	`dict`

Casting and Type

Casting can be used to change/specify the data type of a variable

```
a = int(7.1)   # a = 7
b = str(7)     # b = '7'
c = float(7)   # c = 7.0
d = list(("apple", "banana", "cherry")) #i.e. change a tuple to list
```

We can check the data type of a variable using the `type()` function

```
a = 7
b = "hello"
print (type(a))
print (type(b))
```

Type Annotations (Type Hints)

Python allows you to optionally indicate

- the intended data type of the variable in your code

Examples:

```
name: str = 'Nick'
contact: int = 67904522
```

Type annotation makes the code easier to understand (for others and your future self)

- by explicitly stating the expected types of data

Casting for `input()`

`input()` function by default treats the input it receives as string

- which can be changed using type casting for further manipulation

```
x = float(input("Key in first number"))
y = float(input("Key in second number"))
print(x + y)
```

Inplace Operations

It is common within applications to have code like this:

```
x = x + 1
```

- which can be written in shortcut as an inplace operation:

```
x += 1
```

Any mathematic operator can be used before the '=' character to make an inplace operation

`+=` increment the variable in place
`*=` multiply the variable in place
`/=` divide the variable in place
`%=` return the modulus of the variable in place
`**=` raise to a power in place

String

String Manipulation

String is the data type that we use to store text

- which is the most practical way to store and share information.

Python strings are hence arrays of bytes representing (Unicode) characters

- a string is created by putting a text in (double or single) quotes

```
text = "Hello World!"
```

Square brackets (known as indexing by number) can be used to access elements of strings

```
print(text[2])
```

We can return a range of characters by using the slice syntax to return a part of the string

```
print(text[2:5])
print(text[:5])
print(text[1:])
```

String Manipulation (cont.)

Python provides various functions and methods* that can be used to manipulate string

`len()` function can be used to get length of a string

```
text = "Hello World!"
print(len(text))
```

We can change the case of the string using `upper()` and `lower()` methods

```
print(text.upper())
"Hello World".lower()
```

*method – more on this later

String Manipulation (cont.)

We can check whether there are upper case and lower case in a string using `isupper()` and `islower()` methods

```
text = "Hello World"
print(text.isupper())
print(text.islower())
```

We can search for specific character(s) using the `find()` method

```
text = "Hello World"
print(text.find("Wo")
```

String Manipulation (cont.)

The `strip()` method removes any whitespace from the beginning or the end

```
a = “  Hello World  "
print(a.strip())  #output = “Hello World”
```

We can combine string with number using `format()` method

```
age = 22
txt = "My name is Chen, and I am {}"
print(txt.format(age))
```

There are many other string methods that you can use on strings – see Python references

Conditional Execution

Conditional Execution

Real world practical program will only be useful

- if it can make decision to change the execution based on certain conditions

These conditions can be determined by using the the logical expression

- such as the "`if`" statement

Example:

```
a = 38
b = 24
if a > b:
    print("a is greater than b")
```

The expression `a > b` is known as a logical expression, or Boolean Expression

Logical/Boolean Expression

Logical (or Boolean) expression produces a `True` or `False` result

- which can also be interpreted as number 1 or 0
- used extensively for conditional execution of algorithms

Common examples:

- Equals: `a == b`
- Not Equals: `a != b`
- Less than: `a < b`
- Less than or equal to: `a <= b`
- Greater than: `a > b`
- Greater than or equal to: `a >= b`

if-elif-else

Python uses colon and indentation (whitespace or Tab at the beginning of a line) to define the scope in the code

```
a = 38
b = 24
if a > b:
    print("a is greater than b")
elif a == b:
    print("a and b are equal")
else
    print("a is less than b")
```

colon

Indentation

Multiple conditional statements can be combined using logical **and**

```
a = 200
b = 33
c = 500
if a > b and c > a:
    print("Both conditions are True")
```

`match-case` statement

Python (3.10) also has the **match-case** statement

- can also be used for comparison like if-elif-else
  - but allow for more complicated pattern matching
- reduce amount of code needed and improve the readability

```
character = 'A'
match character:
    case 'A':
        print("character is A")
    case 'B':
        print("character is B")
    case 'C':
        print("character is C")
```

Iteration by looping

Looping is very useful in program

- many practical programs need to do the same thing many times
- Loops repeat execution of certain code hence reduce the amount of code we need to write

Python provide two types of recursion support

- `while` loop
- `for` loop

With the `while` loop

- we can execute a set of statements as long as a condition is true.

```
i = 1
while i < 6:
  print(i)
  i = i + 1
```

Note the Indentation – it defines the scope of the recursion

Iteration – `while` loop

We can also have a infinite loop

- which we can break out of the loop by using the **`break`** statement

```
i = 1
While True:
  print(i)
    if i == 3:
      break
  i += 1
print("done")
```

We can also skip the current iteration and continue with the next iteration using the **`continue`** statement:

```
i = 0
while i < 6:
  i += 1
  if i == 3:
     continue
  print(i)
```

Iteration – `for` loop

Python's for loop is typically used when we know the number of time we want to loop through a set of code

- e.g. together with the `range()` function

```
for x in range(10):  #from 0 to 9, not 10
  print(x)
```

`range()` function generates a sequence of numbers (and return an iterator)

- starts from 0 by default, and increments by 1 (by default)
  - ends at a specified number
- but we can vary the default

```
for x in range(3, 10):  #from 3 to 9
  print(x)

for x in range(2, 30, 3):  #from 2 to 29, increment by 3
  print(x)
```

`for` loop

Python's `for` loop can also be used to iterate over other sequence

- E.g. a list, a tuple, a dictionary, or even a string

```
for x in "apple":
  print(x)
```

The `pass` statement can be used to skip the operation

- e.g. used as a 'placeholder' for future code

```
for x in [0, 1, 2]:
  pass                    #don't do anything
```

for _ Loop

When it is not necessary to use the element returned by the sequence (e.g. range) within the loop body

- we can use the single underscore symbol _ instead

```
for _ in range(3): # use underscore instead of a variable
    print("Hello World!")
print("How are you? today")
```

Summary of Python Basics

- **Basic I/O Operations**
- **Variables**
- **Data types**
- **Conditional Execution: if, while, for**
- **break**
- **continue**
- **pass** statement