Fundamental Computing – Practical Content Review B

- **File I/O**
  - Opening files
    - Opening a file can be done using the open(<filename>, <mode>) inbuilt function
    - <filename> corresponds to the name of the file you wish to access
    - By default, the path is set to that of the .py file; this path may be changed if necessary
    - <mode> may either be "r" (i.e., read); "w" (i.e., write), or; "a", append
    - By default, the mode is set of read
    - The open(…) function will return a file handle that may be manipulated in a read or write fashion that is dependent on the mode specified
    - Typical example usage:
      - Get the read access fileHandle:
        ```
        fileHandle = open("DATA.TXT")
        ```
      - Get the write access fileHandle:
        ```
        fileHandle = open("DATA.TXT", "w")
        ```
      - Get the write access fileHandle:
        ```
        fileHandle = open("DATA.TXT", "a")
        ```
      - The difference between "w" and "a" will be described below
  - Closing file handles
    - Once processing on a file handle is complete, that file handle should be closed via the file handle method close() – i.e., by executing: `fileHandle.close()`
  - Reading from a file
    - Given a read access file handle:
      - Data from the file may be read using the following examples
        - Example 1:
          ```
          fileHandle = open("DATA.TXT")
          data = fileHandle.read()
          fileHandle.close()
          ```
        - Example 2:
          ```
          data = []
          fileHandle = open("DATA.TXT")
          currentLine = fileHandle.readline()
          while currentLine:
                data.append(currentLine)
                currentLine = fileHandle.readline(
          fileHandle.close()
          ```
          - Note: end of the file is reached when fileHandle.readline() returns the empty string
        - Example 3:
          ```
          data = []
          fileHandle = open("DATA.TXT")
          for line in fileHandle:
                data.append(line)
          fileHandle.close()
          ```
  - For the parsing the data within a file, please review string methods (from Content Review A)
  - Writing to a file
    - A write access file handle may be obtained by either specifying the "w" or "a" modes
    - When "w" mode is specified:
      - If the file does not already exist, it will be created; data may then be written to that file
      - If the file already exists, new data written to the file will overwrite any old data in that file

- When "a" mode is specified:
  - If the file does not already exist, it will be created; data may then be written to that file
  - If the file already exists, new data written to the file be appended at the end of the original data in that file
- To create and/or write to a file, the interpreter must have file permission at that location
- Given a write access file handle:
  - Data may be written to the file via the following example
    - Example 1:

```
fileHandle = open("DATA.TXT", "w")
for i in range(len(data)):
      fileHandle.(str(data[i]) + "\n")
fileHandle.close()
```

    - Example 2:

```
fileHandle = open("DATA.TXT", "a")
for i in range(len(data)):
      fileHandle.(str(data[i]) + "\n")
fileHandle.close()
```

  - Note: in the examples above, we assume that data is a collection

- **Exception Handling**
  - To manage exceptions within code, we must utilise a try-except block
  - The general structure of a try-except block is as follows:

```
try:
      # block of code where we expect 1
      # or more exceptions to be raised
except <exception type 1>:
      # code to run when
      # <exception type 1 is encountered>
except <exception type 2>:
      # code to run when
      # <exception type 2 is encountered>
... ... ...
except:
      # code to run when a general exception
      # is raised (but one that is not caught in
      # one of the specific except
      # <exception type i> blocks already expressed
else:
      # code to run when no exceptions
      # are raised in the try block
finally:
      # code that is always run regardless
      # of whether an exception is raised or not
```

  - Note: the try-except block will only account for exceptions encountered within the try block; this means that if an exception is raised because of the code in other parts of the try-except block, those will not be handled unless there are nested try-except blocks – i.e., care should be taken to ensure that code in the non-try blocks inside a try-except block will not be prone to exceptions

- **Object Oriented Programming (OOP)**
  - OOP allows us to conceptualise new data types that represent actual things
    - Examples: shapes, people, dates (days, months, years), nodes in a data structure, etc.
  - To use OOP:
    - Define a class – i.e., a template of an object
      - Example:
```
Class Square():
    # define methods for this class here
```
    - Define the initialisation method for that class – i.e., define the attributes to be stored
      - Example:
```
Class Square():
    def __init__(self, length):
        self.length = length
```
      - Note that this is a special method that will be executed whenever an instance of the specified class is created
      - Also note that methods are simply functions that are specified within a class
      - Instances will be described below, when we describe class instantiation
      - The `self` parameter must always be specified for each class method, this allows an instance to refer to itself
      - "`self.length = length`" is an assignment statement that is applied to an the instance of the class is question – i.e., for the given instance, we now assign it a variable (referred to an attribute in the context of classes), length
      - It should be noted that class attributes may simply be specified within the class
        - Example:
```
Class Square():
    length = None
```
        - Typically, the above is only used when declaring static attributes. For typical attributes, you should use the `__init__` method instead
    - Create an instance of a class
      - Example:
```
x = Square(10)
```
      - Creating an instance requires you to specify the class in a similar manner you should call a function. However, the method run on such instantiation is the `__init__` method with the class instantiated. This is why when a new instance of a class is created, you are required to also input the number of parameters as specified by the __init__ method – e.g., in the above case, `Square.__init__` has 1 input, which is why 1 input is given for the instantiation of x
    - Define other methods in the class
      - Example:
```
def calculateArea(self):
    return self.length ** 2
```
      - As mentioned above, each method defined in a class must include self as its first parameter
      - When such methods are called, they are typically only called on instances of the class
        - Example:
```
x = Square(10)
print(x.calculateArea())
```
      - For each such call, the self parameter allows the interpreter to reference the calling instance – this is done automatically for you (you only need to use the required convention of specifying self as the first parameter of each class method)

- There are several special methods, that when implemented, will overwrite standard Python functionality. These include:

`__str__`	`__lt__`	`__le__`	`__eq__`	`__ne__`
`__gt__`	`__ge__`	`__hash__`	`__bool__`	`__new__`

  Each of the above methods overwrite some special Python functionality. You should explore each in detail if time permits
- Encapsulation and data hiding
  - By creating a class to house attributes and methods pertaining to a single object, encapsulation is achieved
  - Data hiding is in effect when we make the attributes of the class in question private
    - Private attributes (and methods) are achieved by convention only in Python; this is done by adding a single "_" infront of the attribute or method
      - Example:
```
def __init__(self, data):
    self._data = data
```
      - Notice that the attribute stored is `_data`, and not just `data`, which signifies that the attribute is private
      - Private attribute should be accompanied by get and set methods which ensure that their usage and manipulation is regulated
        - Example:
```
def getData(self):
    return self._data
def setData(self, newData):
    self._data = newData
```
        - Essentially, by using such methods, you are able to control how the attribute `data` can be manipulated (e.g., via validation through exception handling, etc.) and stored
- Inheritance
  - When defining a class, you may define it to be a child of another class; doing so would mean that the defined class would inherit all the attributes and methods of the parent class
    - Example:
```
Class Rectangle(Square):
    pass
```
    - In such cases, all attributes and methods defined in the parent class (i.e., Square), will also be accessible in the child class
      - Example:
```
y = Rectangle(10)
print(y.calculateArea()
```
  - Multiple inheritance may also be performed, in which case, Python's Method Resolution Order (MRO) is required; you can read more about this at: https://makina-corpus.com/blog/metier/2014/python-tutorial-understanding-python-mro-class-search-path
  - Thankfully, for you, multiple inheritance is not included in the A-level syllabus, so you may read the above at your leisure

- Polymorphism
  - The polymorphism of child classes is typically achieved by changing the way certain methods in the parent class work; this includes the `__init__` method
  - Polymorphism typically applies to class methods and not attributes. However, a polymorphed class can typically also include attribute attributes since the `__init__`method can be polymorphed – see the example below
  - The most obvious way to do this is to simply redefine a method defined in the parent class
  - More subtly, you can also use the `super()` built-in function to access a method in the parent class, and then add code as required
    - Example:

```
Class Rectangle(Square):
      def __init__(self, newLength, newBreadth):
            super().__init__(newLength)
            self._breadth = newBreadth
```

    - In the above example, `super()` is used to access the parent (i.e., `Square`) class's `__init__` method, which is called with the 1 required argument; this would initialise the _length attribute in any new `Rectangle` instance; subsequently, the `_breath` attribute is initialised (normally)
  - When looking at a UML class diagram, polymorphed methods can typically be spotted by noticing the methods that have been re-defined within the child class