**Fundamental Computing – Practical Content Part A**

* **Primitive Data Types**
  + Integer
  + Float
  + String
  + Boolean
* **Basic Operations on Primitive Data Types**
  + Arithmetic operators

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| + | – | \* | / | % | // | \*\* |

* + Logical operators

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| == | != | < | <= | > | >= | or | and | not | in |

* + String operators

|  |  |
| --- | --- |
| + | \* |

* + Precedence
    - Use brackets (i.e., “(“ and “)”) when unsure
    - Review: <https://docs.python.org/3/reference/expressions.html#operator-precedence>
* **Variables and Assignment**
  + Legal variable names
    - Cannot begin with a digit
    - Cannot include operator symbols
    - Cannot be reserved words (e.g., or, and, not, in, is, def, return, pass, break, continue)
    - Should not be built-in function names (e.g., print, input, range, len, min, max, int, str)
  + Assignment
    - In the form: <variable name> = <expression>
      * <expression>: <operand> <operator> <operand> |

<expression> <operator> <expression>

* + - * <operand>: <literal value> | <variable> | <function call>
      * Examples of literal values: -1, 0, 1, 3.14, “hello”, True, None, etc.
    - Updates the lookup table such that the variable in question now exists, and is associated with a value at the specified memory location
* **Input and Output**
  + Printing to the terminal with the print(...) statement
    - Should always be used as a function call
    - Should always only take in a single string corresponding to the desired output
  + Requesting user input via the input(…) statement
    - Should always be used as part of an assignment statement
    - Should always only take in a single string corresponding to the desired output (i.e., instruction to the user)
* **Type Casting**
  + Converting a value of one type to another (e.g., string to integer); performed by calling:
    - int(…)
    - float(…)
    - str(…)
    - bool(…)
* **Control Structures**
  + if-elif-else statement
    - First conditional execution should be in the form:
      * if <Boolean Value>:

<CODE BLOCK>

* + - Intermediate conditional executions should be in the form:
      * elif <Boolean Value>:

<CODE BLOCK>

* + - Final block should be in the form:
      * else:

<CODE BLOCK>

* + - Note that all blocks in an if-elif-else statement are mutually exclusive (i.e., only 1 of them will be executed)
  + while loop
    - Should be in the form:
      * while <Boolean Value>

<CODE BLOCK>

* + for loop
    - Should be in the form:
      * for <variable> in <collection>

<CODE BLOCK>

* + - * Note that in the case of a for loop, the specified variable will take on each value in <collection> (following the order of the collection) – i.e., for the i-th iteration of the loop, the variable will take on the i-th value in the collection
      * The values of collection that are iterated are assigned prior to the first iteration, and are NOT subject to changes in the specified collection – i.e., even if the collection changes throughout the various iterations, this will not affect the values that the specified variable with iterate through
  + Note that the flow within loops may change by using:
    - break
      * Stops the loop entirely when called; code after the loop continues
    - continue
      * Stops the current iteration of the loop when called; code in the next iteration continues
  + Helpful built-in functions to assist with control structures:
    - range(…)
      * range(a) is a collection (0, 1, 2, …, a)
      * range(a, b) is a collection (a, a+1, a+2, …, b-1)
      * range(a, b, c) is a collection (a, a+c, a+2c, a+3c, …, b-1)
    - len(…)
      * len(L) returns the number of elements in the collection L
* **Strings and Lists**
  + These are both collections
  + Strings are immutable (value cannot be modified)
  + Lists are mutable (value can be modified)
  + Indexing
    - L[i] returns the i-th element of a string or list (starting at index 0)
  + Reverse indexing:
    - L[-1] refereces the last element in L
    - L[-2] references the second last element in L
    - L[-a] references the a-th last element in L (i.e., a elements from the back)
  + Slicing
    - L[a:b] returns the sub-collection (L[a], L[a+1], L[a+2], …, L[b-1])
    - L[a:b:c] returns the sub-collection (L[a], L[a+c], L[a+2c], L[a+3c], …, L[b-1])
    - Shortcuts:
      * L[:] returns all elements in L
      * L[:a] returns all elements in L up to (but not including) index a
      * L[a:] returns all elements from index a till the last element in L
  + Copying lists
    - Since lists are mutable, they are copied by reference; to copy a list, you need to manually copy each element into the new list, or use a slice of all elements
  + List comprehension
    - For example: [x for x in range(100)]
    - Note that there are many complex usages for list comprehension; you should explore these carefully
  + You may casting from list to string and vice versa; be sure to explore how these work
  + Helpful string methods:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| index(…) | lower() | upper() | replace(…) | join(…) | format(…) | isspace() |
| isalnum() | isalpha() | isdigit() | isdecimal() | isnumeric() | islower() | isupper() |

* + - You should review all the above functions and learn how to use them
  + Helpful list methods:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| append(…) | pop(…) | index(…) | reverse() | sort() |

* + - You should review all the above functions and learn how to use them
* **Dictionaries**
  + A special form of list that allows each element index to be a specified value; for example D = {“a”: 1, “b”: 2}, where D[“a”] will return 1 and D[“b”] will return 2
  + Each element of the dictionary thus has 2 parts:
    - Key – the index value to reference if the correspond value is desired
    - Value – the value stored
    - From the example above, “a” and “b” are keys, while 1 and 2 are their corresponding values
  + Helpful Dictionary Methods:

|  |  |
| --- | --- |
| keys(…) | values(…) |

* + - You should review all the above functions and learn how to use them
* **Functions**
  + Utilised to write a specific module or unit of code
  + Defined by using def
    - For example:

def <function name>(<comma-separated parameter list>):

<CODE BLOCK>

* + - A return line will cause the function to stop executing and return the specified value
      * For example:

return True

* + - When no return statement is specified or if a function resolves without a return value, a None value is automatically returned
      * Note that the None value is a special value signifying “no value”
  + Execution of functions utilises what is known as a call stack
    - The most important aspect of the call stack is that each allows each function call to have its own lookup table
    - Thus, unless a variable is specified as global, a function should only utilise local variables (i.e., variables either defined as parameters of the function or within the function itself)
* **Recursion**
  + Instead of using a loop to repeat functionality, uses nested function calls
  + Requires the specification of:
    - Base case(s)
    - Recursive case(s)
  + Wrapper-based recursion may be used when some functionality need only be run once; i.e, this corresponds to the usage of a main calling function that calls the actual recursive function
* **Other Helpful Built-in Functions**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| abs(...) | bin(...) | chr(...) | list(...) | hex(...) | map(...) | min(...) | max(...) |
| oct(...) | ord(...) | pow(...) | round(...) | reversed(...) | sorted(...) | sum(...) | tuple(...) |

* + You should review all the above functions and learn how to use them
* **Helpful Modules**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| random | math | datetime | re | csv |

* + You should review all the above modules and learn how to use them

**Fundamental Computing – Part 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