**Lessons learned reflection**:

* Create a 2-3 page summary that outlines the lessons learned in this Basic Programming course.

##############################################################################

**Module 1**

Software applications are used for many important aspects of our daily lives.  We manage our finances, navigate our vehicles and track our health with software applications.  To ensure that the application is the highest possible quality, structured processes must be defined and followed.  Developing software using proven methodologies maximizes the project success. Experience has shown that an organized approach typically results in a better outcomes.

The stages in this list frame a successful software development lifecycle. (SynapseIndia, n.d.)  The stages are;

1. Planning - Identifying resource needs and appropriate skillsets maximize the effectiveness and efficiency of project  execution.
2. Analysis - Business needs and software functionality must be defined before coding begins. It is critical to know the details of what the application is expected to deliver.
3. Design -  The results of the analysis stage are used to formulate the framework of the solution.
4. Creation/Implementation - The application is written to the design specifications and introduced into a software test environment to be proven-out.
5. Testing/Documentation - Use of the application in a simulated work environment is necessary to determine if the specifications have been met. Testing is performed manually and increasingly with automated test tools.  Documentation is created to enable usability and ensure the code can be maintained more easily in the subsequent stage.
6. Maintenance - Once a completed application is put into production, it requires updates and enhancements to keep up with business needs.

Modern software design has evolved over time, as a result of the development life-cycle.  Program modularity enables teams working on different aspects of an application to work in parallel, speeding development.  This style of programming also results in code modules that can potentially be used in combination with new code modules to create new applications.  Modularity also allows applications to be adapted more easily, as requirements change over time.  Additionally, modular code is easier to read and debug.  All these factors contribute to lower development costs and higher-quality software.

SynapseIndia (n.d.), 6-stages of Software Development.  Retrieved from;

<https://www.synapseindia.com/6-stages-of-software-development-process/141>

**Module 2**

*Data collection and retention is a very common purpose for application software.  Building a simple database for recording data for individual family members requires that the data elements be defined.  There are numerous fields that can be chosen, but I have selected the elements listed below;*

* *Salutation - string*
* *First name - string*
* *Last name - string*
* *Address -  (multiple fields) string*
* *Zip Code - integer*
* *Birthdate - date*
* *Phone number - string*
* *Email address - string*

*Since this hypothetical scenario is much like a database, using a composite data type like a dictionary would be practical.  The dictionary is a collection of objects that can be accessed by associating each element with a key (Sturtz, n.d).  The elements in my list could easily be used as the keys to retrieve the associated values of each dictionary object.  The program code would easy to read and augment, should the number and types of fields need to be adjusted.  A dictionary has many built-in functions that would make the program very capable of processing the contained data.*

Sturtz, J. (n.d). Dictionaries in Python.  Retrieved from: [https://realpython.com/python-dicts/](https://realpython.com/python-dicts/#operators-and-built-in-functions)

**Module 3**

In the mid-1850's, a professor of mathematics at Queen's College, in Cork, Ireland published his best known work, The Laws of Thought. This work, authored by George Boole contained what later came to be known as Boolean Algebra.  This is a branch of algebra where the values of variables are interpreted as either True or False (1 or 0) (Boolean Algebra, n.d.).  Boolean Algebra is a formal manner for describing logical operations, similarly to how elementary algebra describes numeric operations. Due to its binary nature, Boolean logic fits well with digital computer operation.  It is clear that Boole's work has made a major contribution to the advent of the Information age (George Boole, n.d.).

Boolean logic can be used to formulate equations that result in either a positive outcome (True), or negative (False) (Boolean Algebra, n.d.). This approach is used in computer programming languages resulting in code containing complex logical sequences.  There are a number of mathematical operators that express the relationship between equation components.  Here is a chart showing Boolean mathematical operators that are used in programs to compare objects or variables:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| == | Equal to | If (age== 20) |
| != | Not equal to | If (score!= 100) |
| > | Greater than | If(num\_people > 10) |
| < | Less than | If (count < 25) |
| >= | Greater than or equal to | If (total >= 5) |
| <= | Less than or equal to | If (value <= 200) |

(Relational Operators, n. d.)

When working with complex programs, it is possible to determine the outcome of an expression prior to the full execution of the code.  If this occurs, it is not necessary to continue processing the expression once the outcome is already known.  This situation is managed by use of Python shortcut operators.  If an expression evaluates an initial subexpression to be true, it will not use resources to evaluate the second subexpression.  This results in faster execution of the program by not processing unnecessary statements.

Examples of shortcut operations:

# Do something if a flag is False:

if not my\_flag:

    print("Hello!")

# This...

if not x == 5:

    x += 1

# ... is equivalent to this:

if x != 5:

    x += 1

(Relational Operators, n.d.)

George Boole likely never understood that his adaption of human logic to algebraic equations would become a significant pillar in information technology.  Nonetheless, his work has been leveraged to create amazing capabilities in computer languages and their ability to digitize our World.

**References:**

George Boole (n.d.). Retrieved from [https://en.wikipedia.org/wiki/George\_Boole (Links to an external site.)](https://en.wikipedia.org/wiki/George_Boole)

Boolean Algebra (n.d.) Retrieved from [https://en.wikipedia.org/wiki/Boolean\_algebra (Links to an external site.)](https://en.wikipedia.org/wiki/Boolean_algebra)

Relational Operators (n.d.) Retrieved from <https://www.python-textbooks-readthedocs.io/en1.0/Selection-Control-Statements.html>

**Module 4**

Like most programming languages, the Python language offers techniques to repeat a series of program statements.  These statements repeat for the duration of a certain condition or for a specified number of repetitions.  This programming technique is referred to as a loop. Python loops can be written in multiple forms and are highly adaptive to a wide-range of program requirements.

The while loop

The while loop is a commonly used technique for processing repetitive data. The repeated statements in a while loop are called the while clause.  The clause will repeat as long as the statement’s condition is evaluated as True.  A while statement contains: The while keyword and an expression that evaluates to True or False, followed by a colon. The next line starts the while clause, which is an indented block of code. This type of loop is the best choice, if code is to execute for a non-specific number of iterations (Hofmann, 11/17).  This is an

example of a while loop:

spam = 0

while spam < 5:

print('Hello, world.')

spam = spam + 1 (Sweigart, n.d.).

At the end of the first pass of the while clause, the program execution returns the while statement.  The loop evaluates the while condition and the clause is repeated. At the point that the while condition becomes False, the loop is completes and subsequent statements execute (Sweigart, n.d.).

The for loop

The for loop construct uses an iteration value to control the number of repetitions of the loop body.  The iterator  can be a variable that is incremented during each pass through the clause statements.  Or the range() function can be used to set the limits for the number of passes through the clause (Sweigart, n.d.).  The structure of the for loop is otherwise similar to the while loop, in that first line ends with a : and the loop clause statements are an indented block of code (Sweigart, n.d.).  This is an example of a for loop construct:

print('My name is')

for i in range(5):

print('Jimmy Five Times (' + str(i) + ')') (Sweigart, n.d.).

References

Sweigart, Al. Automate the Boring Stuff with Python: Practical Programming for Total Beginners (p. 60). Kindle Edition.

Sweigart, Al. Automate the Boring Stuff with Python: Practical Programming for Total Beginners (pp. 66-67). Kindle Edition.

Hofmann, Frank (11/29/17). Retrieved from <https://stackabuse.com/loops-in-python/>

**Module5**

Good programming practices create code that is efficient and reusable for other projects.  Writing programs with modular functions that perform specific tasks results in programs that are more easily read and understood by others.  Additionally, the 'modules' created by this approach may be used in other programs that require the same functionality.

Effective exchange of code among various projects requires a structured code repository strategy.  A tool like Github or another repository solution can be used for this purpose.  Programmers have created libraries of such custom functions and have shared them internally or publicly.  This allows Python coders to leverage the work of others.  Using shared libraries, programmers can develop an application without having to write every single line of code, themselves.  This approach oftentimes dramatically decreases development time.  In many cases, only small adjustments to 'imported' functions are necessary to adapt them to new projects.

Functions are defined at the beginning of a program and can be 'called' as often as necessary during code execution. Functions work by accepting data from the main body in a systematic manner.  Once data objects are passed, the function performs a process designed to manipulate the data, as required.  When this process completes, the results are 'returned' to the main body of the program for continued processing.

A function block can have parameters defined within their parentheses. Parameters are named entities in the function definition that specify the arguments that can be accepted(Tagliaferri, 2017).  The function add\_numbers in the example below accepts parameters x, y, and z. When the function is called with the framework of the program, values are passed for processing.  The last line in the code block returns the processed values back to the main function (Tagliaferri, 2017).

def add\_numbers(x, y, z):

    a = x + y

    b = x + z

    c = y + z

   return a, b, c

def main():

    sums = add\_numbers(1, 2, 3)

    print(sums)

if \_\_name\_\_ == '\_\_main\_\_':

(Tagliaferri, 2017)

In the example, the processed data is used in a statement that populates the variable 'sums'.  The next line prints the results of the function call.

The use of functions makes program creation more effective and efficient.  By using functions in the body of a program, the resulting code is easier to read and understand.  Additionally, modules are able to be used in other programs thereby, enhancing productivity.

Tagliaferri, Lisa (02/28/2017). Retrieved from <https://www.digitalocean.com/community/tutorials/how-to-define-functions-in-python-3>

**Module 6**

The Python programming language is equipped with the list and dictionary data structures for handling arrays of information.  Though they have similarities in manner they function as data containers, there are a number of key differences in how Python interacts with the structures.

Lists store data elements as a sequence of arbitrary objects.  The objects contained can be of any data type.  Lists are ordered, based-on the order the objects are created. The contents are mutable and can be accessed and changed after a list is created.  Python can iterate over the objects and dynamically interact with the contents. (Difference, n.d.)

Defining and viewing a list:

>>> a = [2, 4, 6, 8]

>>> a

[2, 4, 6, 8]

Appending a list:

>>> a = ['a', 'b']

>>> a.append(123)

>>> a

['a', 'b', 123]

(Sturtz, n.d.)

Lists are defined by square brackets with commas between elements.  The objects are indexed numerically and maintain their order through the lifetime of the list.

Dictionaries also store data in object sequences, but does so in a combination of key/value pairs.  While the values in a dictionary can be any data type, the key must be an immutable type and remain static over the life of the structure.  The keys serve as a index for accessing the data within each object, so dictionaries do not need to maintain the order in which data is entered.

Defining and modifying a dictionary:

>>> dict = {'Name': 'Zara', 'Age': 7, 'Name': 'Manni'}

>>> dict['Age'] = 8;                                                        # update existing entry

>>> dict['School'] = "DPS School";                              # Add new entry

(Python - Dictionary, n.d.)

Dictionaries are bounded by curly brackets and key/value pairs are separated by commas.  The keys are separated from values with a colon.

Lists and dictionaries each have numerous built-in methods that make working with the data contents extremely flexible.  Lists are typically chosen if the data being handled is an ordered collection.  The individual elements are referenced in a linear fashion numerically, beginning with zero.  Dictionaries are a better choice for programmers, when unique data keys are available for use as a  reference for object contents. Dictionaries are capable of locating data very quickly, due to the manner that the keys are mathematically hashed.

Sturtz, John. Lists and Tuples in Python, (n.d). Retrieved from: [https://realpython.com/python-lists-tuples/ (Links to an external site.)](https://realpython.com/python-lists-tuples/)

Difference between list and dictionary, (n.d). Retrieved from: [http://net-informations.com/python/iq/dict.htm (Links to an external site.)](http://net-informations.com/python/iq/dict.htm)

Python - Dictionary, (n.d). Retrieved from: <https://www.tutorialspoint.com/python/python_dictionary.htm>

**Module 7**

Application programs must be able to handle error conditions without crashing and losing data. Syntax and processing exceptions are the two most common error conditions that occur in Python applications.  Fortunately, Python offers the ability to deal with such conditions and allow the program to continue operation, if an error condition occurs.

Parsing error is another term for syntax error.  These come from coding mistakes made during the initial programming phase.  Experience with the language ultimately reduces parsing errors.

Example of Syntax error:

>>> while True print('Hello world')

  File "<stdin>", line 1

    while True print('Hello world')

                   ^

SyntaxError: invalid syntax

(Errors, n.d)

The Python parser aids the determination of the syntax mistake by pointing out the location.  The line number of the program is supplied and an arrow designating the point of the code that was being parsed when the failure occurred. The information contained in the error message is a tremendous help in finding the code that must be corrected, if the program is particularly lengthy.

Code that passes initial inspection of the parser can generate errors for various reasons during program execution.  These include issues such as, illegal mathematical equations and data-type mismatches, resulting from user input.  Such error conditions typically result in a crash and loss of any processing that has occurred, up to that point.

Example of an Exception error:

>>> 10 \* (1/0)

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

ZeroDivisionError: division by zero

(Errors, n.d)

As with syntactical errors, Python generates error messages that aid in identifying the cause of the exception. Information is printed that assists the programmer in identifying causes. This allows the programmer to adjust code to manage or avoid the condition.

A Python code structure tool used to prevent crashes from unexpected conditions is the try: and except: approach. This following is an example of this technique in use;

Example of try: except: technique:

>>> while True:

...     try:

...         x = int(input("Please enter a number: "))

...         break

...     except ValueError:

...         print("Oops!  That was no valid number.  Try again...")

(Errors, n.d)

This technique allows the code in the try: section to execute and if an exception occurs, the error is captured.  This prevents a crash and allows the code following the except: statement to execute.  In the case of the example, the while loop can continue, thus allowing the user to enter a valid value.  When using this technique within a more complex program, successful execution within the try: section causes the except: section to be skipped. The code that follows continues to execute properly.

There are additional techniques to manage error conditions offered by Python.  This includes techniques, such as user-defined exceptions for managing specific and unique code behavior.  Mechanisms like this offer the programmer powerful tools to enable programs to be resilient, when unexpected conditions occur.  Such capabilities contribute to Python's popularity and its status as one of the most popular programming languages currently in use.

Python Documentation, Errors and Exceptions (n.d.). Retrieved from <https://docs.python.org/3/tutorial/errors.html>

**Module 8**