**Python Definitions**

Packages – In Python, a package is a collection of modules. These modules are premade, importable python programs, each made for a specific function. In essence, packages allow a python user to customize their workspace, tailoring it to their expected operations via their collection of available functions and canned routines from the modules within their chosen packages.

Objects – Objects represent the data input into python. These are the variables, be they floats, integers, characters, lists, or something else. Objects are defined and referred to with a name handle, along with a data type.

Functions – Functions are built-in routines, designed to act on a wide range of objects. To execute a function on an object, use the syntax function(object). Like modules, functions are pre-built blocks of Python code, and installing additional modules and packages will make more functions available.

Methods – Methods are a special type of function, designed for a specific object type, which perform a pre-specified operation on the object. Functions are universal, while methods only work on specific object types. Methods are called by putting a . after the object name and putting the name of the method being used after it.

Attributes – Attributes are the inner workings of an object, the definitions and properties behind them. Attributes cover a very broad range of properties, and can be either predefined in the making of an object or function, or they can be completely custom, designed by the user after the fact. The flexibility attributes provide is a key part of object-oriented programming. Like methods, attributes can also be called/defined by placing a . after the object name.

**Lists**

In python, a list is a data structure that is responsible for storing multiple individual objects, or indices. Creating a list requires a specific syntax, with the list assigned a name and each element of the list enclosed in a single set of brackets, separated by commas; an example is shown below.

mylist=[item\_0,item\_1,…,item\_n]

Once a list is made, there are a variety of operations you can perform with them. You can refer to individual elements through indexing (mylist[index]), and a series of elements via slicing (mylist[start:end:jump]). Values can be added to the list by the insert and append methods while they can be removed via the delete function. Operators can also be applied, but caution must be used, as, unless you are operating on individual indices of the same data type, the code will return an error. To operate on an entire list, a loop or list comprehension must be used, and even then, the data types must still match up.

**Indexing and Slicing**

As stated above, indexing allows for an element of a list to be referred to directly. The first element of a list is always indexed as 0, with the last index being n-1, n being the total number of list elements. While indexing only allows for a single value to be specified, slicing allows for a range of values to be extracted from the list, either one-by-one or at larger intervals. When slicing a range, it is important to remember that while the starting index is inclusive of the value you choose, the ending index is not. Therefore, if you are going to the end of the list, you must use the : symbol to grab all of the values, including the last. While indexing gives either a float, int, or char value which operations can work with directly, provided there is no conflict in the data types, slicing keeps the list format, so any attempted operation would still result in an error without a loop or a list comprehension.

**Conditional Statements**

Conditional statements are used to determine whether a statement is true, only executing the code following it if the statement is true. If the statement is false, alternative code may or may not be executed instead. Conditionals always begin with an if statement.

x=4

if x<=10 :

print(x,”is a small number”)

The sample above shows the syntax for a simple conditional statement. Note the colon at the end of the if line, this tells python to indent the next line, putting it inside the conditional loop, only returning if the condition is true. In this case, the print statement will run, since x<=10. Conditional statements can also be applied to multiple variables and conditions, using condition operators (and, or) to link them.

x=14

y=23

if x<=10 and y>x :

print(x,”is a small number, while”,y,”is larger”)

else :

print(x,”is a larger number than either 10 or”,y)

In this case, since x is larger than 10, the first print statement will not run, even though y is larger than x. Instead, the statement following the else line will print out upon running the code. Changing the and condition to or would cause the condition to be met and the first output to be printed.

**For Loops**

A loop directs a script to repeat an operation until an ending condition is met. The most used type of loop is a for loop, and it is useful for working with lists. An example for loop is shown below, along with a description of the syntax.

list=[37,24,85,66,72]

for i in list :

i+=13

print(i)

Running this code would output a series of values based on the operation specified in the loop, and would output, one-by-one, 50, 37, 98, 79, an 85. In this loop, i is not a pre-defined variable. Instead, it is a placeholder, adjusting itself to whatever value the loop is working with in its current iteration. This continues until all elements of the list are operated on. As for syntax, the line structure is similar to that of a conditional statement, with a : necessary at the end of the first line, followed by each line bound by the for statement indented.

**List Comprehensions**

List comprehensions are an alternative to loops for the purpose of working with and manipulating data in lists. Instead of defining a list, and then executing a for loop on it, list comprehension puts the loop and the desired outcome within the initial definition of the list. If the list has predefined values, it still must be defined, but list comprehension can still be performed. An example mirroring the loop example is shown below, and it should produce the same outcome.

list=[37,24,85,66,72]

[i+13 for i in list]

Not only is line space save with list comprehension, it also runs faster than a for loop in python due to the way the language works with each operation. This performance increase is amplified for large datasets. However, list comprehension may be harder to read and harder to implement for more complicated mathematical operations, and so is not always the best option relative to loops.