Intro to Python Cheat Sheet

Objects: An object in python is any (and all) entities that can carry, define, and operate on information. Objects are the variables, lists, and even functions!

Packages: are modules of python code that can augment a user's ability to interact with their data with new/different functions, method, and attributes. Packages are not a part of the main Python framework and thus must be (1) installed into a user's environment and (2) imported into their code when needed.

- Ex. pandas, numpy, earthpy

Functions: are set blocks of code that are called on to output specific data. They are recognized by having parentheses after them. The information contained is the parentheses is referred to as the argument.

- Ex. **print**("Print is a function")

Methods: are functions made designed for specific objects based on object's type. Like functions, they are followed by parentheses to add further data for a desired output. They can be recognized by following an object name with a period placed in between and followed by parentheses, however imported modules/functions use the same format.

- Ex. **Correct:** object_list.insert(1,8); 'object_list' is the object name and 'insert' is the method
- Ex. **Incorrect:** np.mean(array_1); 'np' is the shorthand name of the numpy package, 'mean' is a function specific to np, and 'array_1' is the object name.

Attributes: are internal characteristics of an object. They are specific to each individual object and vary by object data type and can be compared to some objects length, width, height, or mass.

- Ex. my_list.size will output the number of elements contained in the list.
- Ex. array_1.shape will output the size of each dimension in an array.

Intro Python Data Structures:

Lists: are objects that contain a list of *ordered* elements. This means that each element has its assigned place through an associated index. The elements do not need to be of the same data type.

- Ex. my_list = [3, 'j', 52.1]

Lists are also *mutable* which means elements can be added to or removed from the list.

- Ex. my_list.append[2, 'what is an index']
    - My_list will now contain [3, 'j', 'what is an index', 52.1]

Unfortunately, the list's flexibility is significantly diminished when incorporates different data types. When lists contain only integers (e.g., 1, 3, 89) or only floats (e.g., 2.0, 0.842, 1.0003), various operators (e.g., +, -, *) can be employed to manipulate the data they contain. Furthermore, in order to manipulate a list's elements with mathematical operations, each individual element must be called on with a *ForLoop* which will be discussed later.

Arrays: are similar to lists as a data structure but they are better for handling large amounts of data. Like lists, they are *ordered*, *mutable*, and *non-unique*. Arrays are also an example of an object that must be imported into the Python through a package such as *Numpy*. As such, they must be declared when creating them as demonstrated below:

- Ex. my_array = **np.array**([1,2,3,4])
- Ex. my_list = [1,2,3,4]

Depending on the package imported into the script, arrays may or may not be able to incorporate different data types.

A significant advantage of the array is the manner in which is simplifies mathematical operations.

- Ex. exp2 = my_array**2
  print(exp2) => [1 4 9 16]

Indexing: is the ordering of each element based on its location in a data structure. Python uses 0-based indexing meaning the first element in a data structure has an index of 0, the second element has an index of 1, etc. For lists, which are one-dimensional structures, an element is referred to via the following protocol: [start:stop:step].

- Ex. my_list = [6,7,8,9]
  my_list[0:2] => [6, 7]

As seen in this example, the stop index is not included in the output, yet the start index is.

- Ex. my_list[::2] => [6, 8]

The two "empty" colons default to no start/stop indicated, therefore every element is in the list is included. The 2 at the end, however, is the step index that calls every other element that fits the criteria of the list.

Indexing in arrays is slightly more complicated because they have the potential for greater than one dimension. In an array, the start:stop:step format is identical and to call higher dimensional elements, commas are added to specify between row, column, and frame.

- Ex. array1([0:3], [0:2]) will call the first three rows and the first 2 columns contained within.

Interacting with Data Structures

Conditional Statements: are processes that check the state of a program according to chosen criteria. The three conditionals covered thus far are If, Else, and Elif. An *If statement* sets up an action to be taken when a chosen condition is met. *Else statements* can be added to set up a different action when the *if* condition isn't met. On the other hand, an *elif statement* can be used to check for a subsequent state when the *if* condition isn't met.

```
hunger = 2.9

if 8 <= hunger <= 10:
    print("Eat a meal!")
elif 3 <= hunger < 8:
```

```
    print("Eat a snack!")
else:
    print("Remember to hydrate.")
```

Because 'hunger' was initially set to a value of 8, the output would be "Eat a meal!"

For Loops: if one wanted to apply the above state checks to a large number of elements, one would have to run each response through the conditional statement one at a time. Or one could use a *for loop* and check the state of every element in a list.

```
hunger_list = [1,6,3,7,8,4,2,9,10]

for i in hunger_list:
    if 8 <= i <= 10:
        print("Eat a meal!")
    elif 3 <= i < 8:
        print("Eat a snack!")
    else:
        print("Remember to hydrate.")
```

Which would produce the following output:

```
Remember to hydrate. Eat a snack! Eat a snack! Eat a snack! Eat a meal! Eat a snack! Remember to hydrate. Eat a meal! Eat a meal!
```

List Comprehensions: succinctly create new lists based on elements from an existing list. List comprehensions are not the only way to build new lists from old lists—a *for loop* can do the same thing—but they do so with less code.

For example: If the goal was to understand the average level of hunger for anyone needing a snack or a meal in the *hunger_list*, a list comprehension could build a new list with the relevant data very easily:

```
# List comprehension
hunger_list = [1,6,3,7,8,4,2,9,10]

new_hunger_list = [i for i in hunger_list if i >= 3]

print(np.round(np.mean(new_hunger_list), 2))
```

Which would produce: 6.71. Perhaps if you were a camp counselor trying to gauge whether it was dinnertime for the kids or not, you'd be able to tell they can wait a little longer before spaghetti and meatballs.