Intermediate Python

Data Structures: Looping, Lists, Tuples, Dictionaries

Author: W.P.G.Peterson

Assignment Contents:

EXPECTED TIME: 3.5 HRS

Overview

This assignment introduces and explores the fundamental `Python` collection types. It also extends upon topics introduced in the "Basic Python" assignment, and introduces useful `Python` functions and features.

Comfort with the material in these two assignments should translate into a basic comfort with the fundamentals of `Python`. The assignments should also be useful as resources for review. The syntax of `Python` is not particularly complicated, and it is frequently forgiving. However, un-complicated does not mean easy - combination locks are un-complicated but are virtually impossible to open without the combination. Nothing substitutes for practice when it comes to remembering how all these concepts, syntax, and formatting connect.

While the concepts in this assignment are still mostly straight-forward, increasing comfort with this format of assesment is assumed, so expect questions of increasing complexity.

Activities in this Assignment

- Use `for` and `while` loops
- Use lists and list methods
- Review edge cases of variable assignment / manipulation
- Use tuples, zip, enumerate and unpacking
- Use and build non-trivial dictionaries
- Use list/dict comprehensions

Looping Structures

Before heading into the meat of this lesson - collections - we will review and test the use of looping (iteration) structures. Iteration will be a fundamental part of how we test understanding of various collections.

The two looping structures available in Python are `for` loops and `while` loops.

```
### Two sample "for" loops

twos = 2
for num in [1,2,3,4]:
    print(num)
    twos *= 2
print("twos = ", twos)

print("\nLooping through string 'hello'")
for char in "hello":
    print(char)
```

`for` loops execute a set number of times. The syntax is:

- `for <variable name(s)> in <iterable>:`
- then the indented code block.

In general, anything that can be traversed is iterable. For example, lists (as seen above of `[1,2,3,4]`) and strings are both iterable.

One function used in the following questions and introduced in lecture is `range()`.
See examples below:

```
print(range(5))
print(list(range(5)))
print(list(range(3,7)))
print(list(range(0,5,2)))
for n in range(5):
    print(n)
```

Note from the above:

- Printing out a "range" object directly will simply yeild the statement `range(n1,n2)`.
  - Range is a `generator`. A generator provides items (numbers in this case) one at a time.

- `range()` is iterable and useable in `for` loops
    - trying to print out a `range()` object directly does not yield all those numbers.
- `range()` goes *up to but does not include* the stopping number -- like indexing strings.

Question 1:

```
### GRADED
### Example
### Code a function called 'first_div_16'
### ACCEPT two positive integers, n1 and n2, as inputs
### RETURN the first number in range(n1,n2) that is divisible by 16.
### HOWEVER, if no number in the range is divisible by 16 RETURN 0

### YOUR ANSWER BELOW

def first_div_16( n1, n2):
    ### Create for loop with the range
    for n in range(n1, n2):
        ### check if n is divisible by 16
        if n % 16 == 0:
            return n
    ### if none of the numbers divisible by 16, return 0
    return 0

### Testing
print(first_div_16(2,5))
print(first_div_16(2,30))
print(first_div_16(17,32))
print(first_div_16(1,100))
```

Question 2:

```
### GRADED
### Build a function called 'first_starting_vowel'
### ACCEPT a list of strings as input
### RETURN the first string that starts with a lowercase vowel ("a","e","i","o",or "u")
### HOWEVER if no string starts with vowel, RETURN the empty string ("")

### YOUR ANSWER BELOW


def first_starting_vowel(string_list):
    """
    Return the first string in the list that starts with a lowercase vowel

    Positional Argument:
        string_list -- a list of strings

    Example:
        example_list = ["hello","these","are","strings","in","a","list"]
        print(first_starting_vowel(example_list)) # --> "are"
        print(first_starting_vowel(example_list[:2]))#--> ""
    """

    # Go through each string in the passed list
    for string in string_list:
        # if the first character is a lower-case vowel, return
        if string[0] in "aeiou":
            return string

    # If nothing was returned in loop, return empty string
    return ""
```

While Loops:

`while` loops evaluate a boolean condition, and *while* that condition is true, the loop continues to execute.
Whether or not the condition is true is only checked at the end/beginning of each loop.

```
x = 1
while x<10:
    x +=1
    if x %2 ==0:
        print(x)

while True:
    print("\nWill this go on forever?")
    print("It could.")
    break # But now it won't.
```

Although the above boolean condition of `True` will always be true, (and could easily result in an infinite loop) there are a multiple ways of exiting a loop. One is a `return` statement - as seen in our functions - and the other is a `break` statement - as above.

`pass` and `continue` may also be helpful when your loop-logic becomes more complicated. Documentation

Question 3:

```
### GRADED
### Code a function called 'halve_to_2'
### ACCEPT one numeric input.

### If the number <= 0, RETURN -1
### If the number > 0, divide that integer over-and-over by 2 until it becomes smaller than 2.
```

```
### RETURN that smaller-than-2 number

### e.g. input of 4 Will yield 1 (4->2->1), 5 yields 1.25 (5->2.5->1.25) etc.

### YOUR ANSWER BELOW

def halve_to_2( num ):
    """
    Divide input-number by 2 until it becomes smaller than 2, then return.
    If input-number <=0 return -1

    Positional Argument:
        num -- numeric input

    Example:
        print(halve_to_2(4)) #--> 1
        print(halve_to_2(-39)) #--> -1
        print(halve_to_2)(5673) #--> 1.385009765625

    """
    # If number less than or equal to 0, return -1
    if num <=0:
        return -1
    # Otherwise
    else:
        # While the number is greater than or equal to two,
        while num >=2:
            # divide by two
            num /=2
        return num
```

Question 4:

```
### GRADED
### Code a function called 'string_exansion'
### ACCEPT a non-empty string as input
### RETURN a string that contains every other character, 2n+2 times, where n is the original index of the letter.

### e.g. Input of "Hello" should result in "HHlllllloooooooooo".
### Input of "ROBErt" should result in "RRBBBBBBrrrrrrrrrr"

### YOUR ANSWER BELOW

def string_expansion( input_string ):
    """
    Given a string input, return a string containing every other character 2n+2 times
    Where "n" is the 0-based index of the letter

    Positional Argument:
        input_string -- a non-empty string

    Example:
    str1 = "Hello"
    str2 = "naME"

    print(string_expansion(str1)) #--> HHlllllloooooooooo
    print(string_expansion(str2)) #--> nnMMMMMM
    """
    # Start with empty string
    toRet = ""

    # Take every other letter
    s = input_string[::2]

    # Go through the every-other string
    for i, c in enumerate(s):

        # determine the number of times character should be printed
        # Note the index will be halved at this point
        n = 2*((i*2)+1)
        # add that character that number of times
        # to the string to be returned
        toRet += c*n

    return toRet
```

Collections

Cheat Sheet:

Lists: `[]`, mutable, orderd
Tuples: `()`, immutable, ordered
Sets: `{}`, mutable, unordered, unique values -- Not Covered
Dicts: `{:}`, mutable, unique "keys" required

Lists

Last week's assignment used `lists` in a few problems, but they were not fully introduced.

Lists are a collection of items that may be changed (mutable); but do not all need to be of the same type:

```
l1 = [1,2,3,4,5]
l2 = ["1",2,3,"rabbit", False, 4,[1,2,3]]

print(l1, type(l1), len(l1))
print(l2, type(l2), len(l2))
```

Indeed, as shown above, an element of a list can indeed be another list - frequently referred to as nested lists.

Indexing in `lists` is similar to that in `str`ings

```
print(l1[0])
print(l2[::2])
print(l2[-1][1:])
```

Below, a few examples of list methods:

```
l1 = [1,2,3,4,5]
print(l1)

print("\n.append(5)")
l1.append(5)
print(l1)

print("\n.pop(2)")
l1.pop(2)
print(l1)

print("\n.count(5)", l1.count(5))
```

Question 5:

```
### GRADED
### Code a function called 'item_count_from_index'
### ACCEPT two inputs, a list and an integer-index
### RETURN a count (number) of how many times the item at that index appears in the list.

### HOWEVER, if the integer-index is out of bounds for the list RETURN the empty string ("")
### (e.g. list of 3 items, index of 5 is out of bounds)

### YOUR ANSWER BELOW

def item_count_from_index( input_list, index):
    """
    Return the count of items in a list found at a certain index
    If index out of bounds, RETURN ""

    Positional Argument:
        input_list -- a list of items, of unspecified types,
            assume items are comparable. e.g. support == and != comparison
        index -- an integer index

    Examples:
        print(item_count_from_index([1,2,2,3,3,2,4],2)) #--> 3
        print(item_count_from_index([],2)) #--> ""

    """
    # Check to see if index is valid for the list
    # if not, return empty string.
    if (len(input_list)-1 < index) or (index < 0):
        return ""
    else:
        # Otherwise, count the number of times
        # object at index appears in list
        return input_list.count(input_list[index])
```

Question 6:

```
### GRADED
### Code a function called 'length_times_largest'
### ACCEPT a list as input
### RETURN the length of the list times the largest integer (not float) in the list
### HOWEVER if the list does not contain an integer, RETURN the empty string ("")

### YOUR ANSWER BELOW

def length_times_largest(input_list):
    """
    Given a list of objects, return the length of the list times the
    largest integer in the list.

    Positional Argument:
        input_list -- a list of objects of unspecified types.

    Example:
        print(length_times_largest([1,2,3,4])) #--> 16
        print(length_times_largest(["a","b","c",4])) #--> 16
        print(length_times_largest(["1","100",2])) #--> 6
        print(length_times_largest(["a","b"])) #--> ""
        print(length_times_largest([0.0,40.6])) #--> ""
    """
    # Create Variable for "largest"
    big = ""

    # For each elment in the input
    for e in input_list:

        # If it is an int
        if type(e) == int:

            # If no largest int has been found yet,
            # Assign to `big`
            if big == "":
                big = e

            # If the element is larger than `big`, reassign
            elif e > big:
                big = e
```

```
            # Otherwise, do nothing.
            else:
                pass

    # After going through each element, if no integer found,
    # return empty string.
    # Otherwise, perform calculation and return
    if big == '':
        return big
    else:
        return len(l)*big
```

Question 7:

```
### GRADED
### Code a function called 'combine'
### ACCEPT two inputs:
### The first input is a list.
### The second input is either either a list or some other type of object

### IF AND ONLY IF the second input is a *list*;
### ### "extend" the first list by adding to it the elements of the second list.
### e.g. if the inputs are [1,2,3], [4,5], the output should be [1,2,3,4,5] NOT [1,2,3,[4,5]].

### IF the second input is NOT a list, append that item to the original list.
### e.g. if the inputs are [1,2,3], (4,5), the output should be [1,2,3,(4,5)] NOT [1,2,3,4,5].

### RETURN the resulting combination.

### YOUR ANSWER BELOW

def combine(list1, to_add):
    """
    Return the combination of the two inputs

    Positional Argument:
        list1 -- a list of objects
        to_add -- an object, list or otherwise

    Example:

        l1 = [1,2,3]
        a1 = [4,5]
        a2 = "b"
        a3 = (2,"b") # a tuple

        print(combine(l1,a1)) #-->[1,2,3,4,5]
        print(combine(l1,a2)) #-->[1,2,3,'b']
        print(combine(l1,a3)) #-->[1,2,3,(2,'b')]
    """
    # If the second argument is a list, extend
    # otherwise, append
    if type(l2) == list:
        l1.extend(l2)
    else:
        l1.append(l2)

    return l1
```

Variable Assignment

In a number of the above exercises you should have used some list methods in your solutions. These can potentially cause problems depending on how variables are defined.

In the last assignment, I stated that **most** of the time, when variables are defined by being set equal to another variable, a change in one variable will not affect the other variable.

The below will show some cases when that is not true.

```
a = [1,2,3]
b = a
print("a: ", a)
print("b: ", b, "\n")

print("Appending 4 to a, and only a")

a.append(4)
print("a: ", a)
print("b: ", b, "\n")

print("setting a[0] = 0")
a[0] = 0
print("a: ", a)
print("b: ", b, "\n")

print("setting a = [1,2,3]")
a = [1,2,3]
print("a: ", a)
print("b: ", b, "\n")
```

In general, when changing the value of a variable using `"="`, other variables will not be changed. However, when the values associated with variables start to change *in-place* there might be trouble.

Python "variables" are actually just pointers that direct the interpreter to a place in memory. When a variable takes on a new value either:

1. The value at the specific place in memory changed.
2. A new memory space was allocated with a new value in it, and the pointer changed to direct to that new location.

```
b = a
# Print out the memory location of each variable using `id()`
```

```
print(id(a))
print(id(b))
```

Question 8:

```
### GRADED
### In the above example. When `b` is set equal to `a` <b = a>, in how many locations is Python storing the list
### [1,2,3]?

### 'a') 1
### 'b') 2
### 'c') Python isn't storing it; Jupyter is.

### Assign the string associated with your choice to ans1
### YOUR ANSWER BELOW

ans1 = 'a'
```

Question 9:

```
### GRADED
### Assume once again b is set equal to a as in the above question.
### Which of the following COULD cause a change in value to both a and b

### 'a') the use of a list method
### 'b') the deletion of one of the variables <del(<var>)>
### 'c') defining a value for one of the variables using '='
### 'd') the setting of a value in the list using indexing

### pick all that apply in a list of string: e.g. ["x","y", "z"] and assign to ans1
### if none apply, return empty list "[]"
### YOUR ANSWER BELOW

ans1 = ['a','d']
```

Tuples

Pronounced either "tup-" or "toop- / tewp-"

Tuples are similar to lists; they are ordered, they can hold variables of multiple types, they are indexable and iterable.

Their major difference -- Tuples are immutable. Once a tuple has been created, it cannot be changed.

```
my_tup = (1,2,3)
my_tup[2] = 4 # Error
```

Thus, changing a tuple requires destruction and creation.
In many cases lists could be used instead of tuples, however, Tuples do help to ensure that the values, or number of values are not changed in a particular collection.
This might be useful when passing collections of parameters between functions, or for ensuring a collection has a certain length. For example if you want to pass an object and a calculated attribute around together, such as a word and it's length, ('word', 4), might be a better choice than ['word',4].
Ultimately tuples can offer safe-guards in your programming when changing values (mutability) is not desired.

Question 10

```
### GRADED
### Code a function called 'type_and_length'
### ACCEPT one input of any of the six types used thus far (int, str, float, bool, list, tuple)
### RETURN a 3-tuple of (input, type, length)

### Hint: type can be found with type(<input>)
### If the <input> does not have `len()` return None for length.
### Note, None is not a string, it is a type and object in and of itself.

print(type(None)) # None example

### YOUR ANSWER BELOW

def type_and_length(obj):
    """
    Return a tuple containing the inputted object, its type and length

    Positional Argument:
        obj - an object that may be any of the following:
            (int, str, float, bool, list, tuple)

    Example:
        for obj in (1,"hi",1.5, True, [1,2,3],(4,5)):
            print(type_and_length(obj))
            #-->
                (1, <class 'int'>, None)
                ('hi', <class 'str'>, 2)
                (1.5, <class 'float'>, None)
                (True, <class 'bool'>, None)
                ([1, 2, 3], <class 'list'>, 3)
                ((4, 5), <class 'tuple'>, 2)
    """
    # Find type of passed object
    t = type(obj)

    # If type can be passed to "len", find length
    # Otherwise, assign "None" to l variable
    if t in [list, str, tuple]:
        l = len(obj)
    else:
```

```
        l = None

    return (obj, t, l)
```

Iterable Unpacking / Zip & Enumerate function

The "unpacking" of variables from an iterable - frequently called "tuple" unpacking becuase it is usually used with tuples - is the ability to assign multiple variables to the items contained in an iterable object.

```
print('tuple')
a,b = (1,2)
print(a,b)

print('\nlist')
a,b = [1,2]
print(a,b)

print('\nrange')
a,b,c = range(3)
print(a,b,c)
```

One final feature of tuple unpacking notation before moving on:
When unpacking, if not all of the values are desired, "the rest" of the values can be dumped into a variable by decorating it with a `*` --- Shown below:

```
a,*b,c,d = range(7)
print(a,b,c,d)

a,b,c,*d = range(7)
print(a,b,c,d)
```

One of the times I most frequently use tuple unpacking is in conjunction with the `zip()` and `enumerate()` functions.

The following demonstration and exercise will expose the behavior of `zip()`

```
z_obj = zip([1,2],[3,4], [5,6])
print(z_obj)

for z in z_obj:
    print(z)

print("\n",z_obj)
print("second for loop using same zip object:")
for z in z_obj:
    print(z)
print("\nend second for loop.")
print("Note: zip is a one-time-use generator")
```

Question 11

```
### GRADED
### Code a function called 'reverse_zip'
### ACCEPT a zip object
### ### The zip object will be in the exact same format as the above example (different values)

### RETURN a list of the 3 lists passed to 'zip()'
### e.g. above the function would return [[1,2],[3,4],[5,6]] for z_obj above

### YOUR ANSWER BELOW

def reverse_zip(zip_obj):
    """
    Given a zip_object, return the lists passed into `zip()` to create
    That zip_object

    Positional Argument:
        zip_obj -- a zip object

    Example:
        zip_obj = zip([5,6],[9,10],["a","b"])

        print(reverse_zip(zip_obj)) #--> [[5,6],[9,10],["a","b"]]
    """
    # Create three lists to populate
    l1 = []
    l2 = []
    l3 = []

    # Pull out objects from zip one-by-one
    for obj in zip_obj:

        # Add elements from objects to lists in order
        for l, o in zip([l1,l2,l3],obj):
            l.append(o)

    # Combine lists into list; return
    return [l1,l2,l3]
```

Like `zip()`, `enumerate()`, creates a generator function that returns tuples, this time with an index and an object:

```
for n, l in enumerate(['a', 'b', 'c', 'd']):
    print(n,l)
```

Enumerate is particularly useful when looping over a list to collect indicies corresponding to particular values.

Question 12

```
### GRADED
### Code a function called 'obj_indicies'
### ACCEPT Two Inputs: a list, then some object to search for.
### RETURN a list of the indicies where that object appears in the list.

### e.g. ([1,2,2,3], 2) should return [1,2]; ([1,2,2,3], 4) should return []

### YOUR ANSWER BELOW

def obj_indicies(list_to_search, search_for):
    """
    Return a list of indicies which are all the locations of the specified
    object in the passed list

    Positional Arguments:
        list_to_search - a list of objects
        search_for - the object to be searched for.

    Example:
        l1 = [1,2,2,3,4,5,10]
        print(obj_indicies(l1,2)) #--> [1,2]
        print(obj_indicies(l1,6)) #-->[]
        print(obj_indicies(l1,10)) #-->[6]

    """
    # Create empty list for indicies
    toRet = []

    # loop through list,
    # if element is equal to what is searched for,
    # add index to list
    for i, e in enumerate(list_to_search):
        if e == search_for:
            toRet.append(i)

    # return list of indicies
    return toRet
```

Dictionaries

At their base, dictionaries are a collection of `key / value` pairs. [See also `"mapping"` and `"associative array"`.]

All the keys in a dictionary must be unique, though they may be many different kinds of objects. Values may also be any kind of object, but values may repeat.

Within a dictionary, the keys and values are associated with `:` inside the dictionary's outer wrapper of `{}`.
e.g `{key1:value1, key2:value2, ...}`

```
### Dictionary with employee ID as key, and name as value
emp_dict = {10024:"John", 10023:"Alice", 105:"Mary"}
print(emp_dict)

print("\ntrying to index dict ordinally")
print(emp_dict[1]) # Error
```

Even though the dictionary "appears" to have an order when it is printed out, in Python's mind, it is unordered. This means two things.

1. Dictionaries are very fast.
2. Indexing occurs using keys

```
# indexing with a key
emp_dict[10024]
```

Despite the fact dictionaries are not orderd, their keys can be looped through.

```
print('\nLooping through dict: ')
for k in emp_dict:
    print("\nkey: ", k)
    print("value: ", emp_dict[k])
```

Question 13

```
### GRADED
### Code a function called "return_value"
### ACCEPT two inputs: a dictionary and a key from that dictionary
### ### ASSUME the provided key is in the dictionary
### RETURN the value associated with that key in the dictionary

### YOUR ANSWER BELOW

def return_value(input_dict, input_key):
    """
    Return the value from the inputted dictionary located at the location of
    the given key

    Positional Arguments:
        input_dict - a dictionary
        input_key - a key in that dictionary

    Example:
```

```
        test_dict = {1:2, "A":"B", 3:"c", "1":"a"}

        print(return_value(test_dict,"A")) #--> "B"
        print(return_value(test_dict,3)) #--> "c"
        print(return_value(test_dict,"1")) #--> "a"

    """
    # Use key accession to return value
    return input_dict[input_key]
```

The above examples are trivial. How ever, w hen you look for them, dictionaries, and dictionary-like object are everyw here. One major example is JSON (JavaScript Object Notataion). Similarly non-relational data-bases w ill use dictionary-like notation. In these complex examples, values are frequently dictionaries themselves.

The below example is from json.org/example

```
tough = {
    "glossary": {
        "title": "example glossary",
        "GlossDiv": {
            "title": "S",
            "GlossList": {
                "GlossEntry": {
                    "ID": "SGML",
                    "SortAs": "SGML",
                    "GlossTerm": "Standard Generalized Markup Language",
                    "Acronym": "SGML",
                    "Abbrev": "ISO 8879:1986",
                    "GlossDef": {
                        "para": "A meta-markup language, used to create markup languages such as DocBook.",
                        "GlossSeeAlso": ["GML", "XML"]
                    },
                    "GlossSee": "markup"
                }
            }
        }
    }
}
```

Suppose you w anted to navigate to the information in line 17 - the key/value pair of "GlossSee":"markup"

```
print(list(tough.keys()))
print(list(tough['glossary'].keys()))
print(list(tough['glossary']['GlossDiv'].keys()))
print(list(tough['glossary']['GlossDiv']['GlossList'].keys()))
print(list(tough['glossary']['GlossDiv']['GlossList']['GlossEntry'].keys()))
```

Only after navigating through four keys, our desired key - "GlossSee" - is exposed.

This complexity becomes a blocker for many people, and they w ill choose to stay aw ay from dictionaries. How ever, particularly w hen progressively building and collecting data, dictionaries offer a fast, native, and organized system for doing so. Similarly, because of their ubiquity, many packages support translation of dictionaries into their particular object type.

Question 14

```
### GRADED
### This exercise involves building a non-trivial dictionary.
### The subject is books.

### The key for each book is its title
### The value associated with that key is a dictionary

### ### In that dictionary there will be Three keys: They are all strings, they are:
### ### "Pages", "Author", "Publisher"

### ### ### "Pages" is associated with one value - an int

### ### ### "Author is associated with a dictionary as value
### ### ### ### That "Author" dictionary has two keys: "First", and "Last" each with a string value

### ### ### "Publisher" is associated with a dictionary as value
### ### ### ### That "Publisher" dict has one key "Location" with a string as value.

### An Example might look like:
### {"Harry Potter": {"Pages":200, "Author":{"First":"J.K", "Last":"Rowling"}, "Publisher":{"Location":"NYC"}},
###  "Fear and Lothing in Las Vegas": { ...}}

### Code a function called "build_book_dict"
### ACCEPT five inputs, all lists of n-length
### ### A list of titles, pages, first<name>, last<name>, and <publisher>location.

### RETURN a dictionary as described above.
### Keys must be spelled just as they appear above - correctly and capitalized.

### YOUR ANSWER BELOW

def build_book_dict(titles, pages, firsts, lasts, locations):
    """
    Return a nested dictionary storing information about Books

    Positional Arguments:
        titles -- A list of strings
        pages -- A list of ints
        firsts -- A list of strings
        lasts -- A list of strings
        locations -- A list of strings

    Example:
        titles = ["Harry Potter", "Fear and Lothing in Las Vegas"]
        pages = [200, 350]
        firsts = ["J.K.", "Hunter"]
```

```
        lasts = ["Rowling", "Thompson"]
        locations = ["NYC", "Aspen"]

        book_dict = build_book_dict(titles, pages, firsts, lasts, locations)
        print(book_dict) # -->
            {'Fear and Lothing in Las Vegas': {'Publisher': {'Location': 'Aspen'},
            'Author': {'Last': 'Thompson', 'First': 'Hunter'}, 'Pages': 350},
            'Harry Potter': {'Publisher': {'Location': 'NYC'},
            'Author': {'Last': 'Rowling', 'First': 'J.K.'}, 'Pages': 200}}

    """

    # Create new dictionary
    toRet = {}

    # zip lists together, pulling out elements one at a time
    for t, p, f, la, lo in zip(titles,pages,firsts,lasts,locations):

        # Build new dictionaries, and associate keys with values
        toRet[t] = {}
        toRet[t]['Pages'] = p
        toRet[t]['Author'] = {'First':f, 'Last':la}
        toRet[t]['Publisher'] = {"Location":lo}

    return toRet
```

Passing Parameters

One final utility of Dictionaries is their ability to to pass parameters into functions. While we have yet to work with complex functions, Machine Learning, for example, calls functions with many many parameters.

```
def example_func(a,b,param2,thatParam):
    print(a,b,param2, thatParam)

params = {"a" :5, "thatParam":"tp", "b":6, "param2":None}

example_func(**params)
```

The above is an example of passing "keyword arguments" to a function [frequently shortened to "kwargs" in documentation]. The dictionary must have keys of strings that match the parameter names, and their associated values will then be placed into the function. To enable this unpacking, the `<**>` is put before the dictionary.

Similarly, parameters of functions may also be passed in via list decorated with a single `*` with the values in correct order.

```
param_list = [1,2,5,6]
example_func(*param_list)
```

Parameters may be passed in via location and by name. **HOWEVER**, named parameters must always come after the location parameters. Once parameters start to be named, they may be put in any order. This is true whether input is by list/dict or conventionally

```
example_func(1,2,thatParam = 6, param2=5)
```

```
l = [1,2]
p_dict = {"thatParam":6, "param2":10}
example_func(*l, **p_dict)
```

```
### Throws SyntaxError
example_func(**p_dict, *l)
```

Question 15

```
### GRADED
### In which of the following is the "*" NOT used in Python?
### 'a') multiplication
### 'b') exponents
### 'c') tuple unpacking
### 'd') passing dictionaries into functions as parameters
### 'e') if statement syntax
### 'f') passing lists into functions for parameter

### Assign string associated with your choice to ans1
### YOUR ANSWER BELOW

ans1 ='e'
```

Dictionary / List Comprehensions

A quick, effective and easy way of building lists or dictionaries is using "comprehensions". Comprehensions incorporate one (or more) for loops within the creation of a list.

```
list_1 = [1,2,3,4]
list_2 = [n*2 for n in list_1]
print(list_2)
```

Comprehensions can also incorporate one or more if/else clauses (elif is not used in comprehensions) either as filters or as switches:

```
l1 = [2,4,6,8,10,12,14,16,18]

# if as a filter:
# Only adding to list if conditional is true
l2 = [n*2 for n in l1 if n%4 == 0]
print(l2)

# if/else as switch
# Always adding to list, if/else determines how it is added
l3 = [n*4 if n % 8 ==0 else n*2 if n% 4 ==0 else n for n in l1 ]
print(l3)
```

As shown above, when filtering in list comprehensions, the `<if>` statment comes after the `<for>` statement. When using the `<if/else/...>` as a switch, the statements come before the `<for>` statment.

Dictionaries can be constructed in a similar manner.

```
string1 = "This is a list of unique strings, said the quick brown fox."
list1 = string1.split(" ")

### ensure there are no duplicate strings by using a `set`
list1 = list(set(list1))

### Dictionary comprehension
dict1 = {s : len(s) if len(s) % 2 == 0 else s.upper() for s in list1}
print(dict1)
```

As shown above, dictionaries are useful when the result of a specific function or functions (particularly if they are long-running functions) need to be associated with a particular object. Again if/[else] clauses can be used as a filter / switch.

Question 16

```
### GRADED
### Code a function called "divisible_by_3"
### ACCEPT a list of numbers (int or float) as input
### RETURN a list of all the numbers in that list that are divisible by 3, multiplied by 2.
### e.g. [1,2,3,4,5,6] as input should yield [6,12]

### ### Preferably use a list comprehension

### YOUR ANSWER BELOW


def divisible_by_3(input_list):
    """
    Return all of the numbers that were divisible by 3 multiplied by 2

    Positional Arguments:
        input_list -- A list of numbers, floats or ints

    Example:
        num_list = [1,4,5,6,7,8,2,9,3,3,6,9]
        print(divisible_by_3(num_list)) #--> [12, 18, 6, 6, 12, 18]

    """
    return [n*2 for n in input_list if n%3 ==0]
```

Review

Question 17

```
### GRADED
### Code a function called "final_element"
### ACCEPT a nested list as an input.
### Input will be in the format [[1,2,3,...,n],[4,5,6,...,n],...,['x','y','z',...,n-1,'q']]
### More specifically, "m" lists, all of "n" elements, will be contained in the nested list

### RETURN the final element in the final list from that nested list.
### In the above example, that would be the string 'q'.

### YOUR ANSWER BELOW

def final_element(input_nested_list):
    """
    Return the final element in the final list of the input

    Positional Argument:
        input_nested_list - a list containing "m" lists,
            each of which has "n" elements

    Example:
        nested = [[1,2,3],[4,5,6],[7,8,"a"]]
        print(final_element(nested)) #--> 'a'

    """
    # access final list
    fin_list = input_nested_list[-1]

    # access final element in that list
    fin_ele = fin_list [-1]

    return fin_ele
```

Question 18

```
### GRADED
### A tuple is:
### 'a') immutable
### 'b') mutable

### assign string associated with your choice to ans1
### YOUR ANSWER BELOW

ans1 = 'a'
```

Question 19

```
### GRADED
### Assume `my_dict` is a dictionary.
### True or False:
### my_dict[0] will return the first element of `my_dict

### Assign boolean choice to ans1
### YOUR ANSWER BELOW

ans1 = False
```

Question 20

```
### GRADED
### True or False:
### A key can be used multiple times in a dictionary.

### assign boolean choice to ans1
### YOUR ANSWER BELOW

ans1 = False
```

Passing Functions

For a final demonstration, I w ant to look at the nature of functions.

Like ints, strings, lists, dictionarys, and everything else in `Python`, funcions are objects. This means functions can be passed around like other objects, and can also be called afterw ards.

The programming exercise "FizzBuzz" asks the follow ing: Given a list of numbers, if a number is divisible by 3 print 'Fizz'. If a number is divizible by 5, print 'Buzz'. If it is divisible by both 3 and 5, print 'FizzBuzz'.

Below I have defined tw o trivial functions that return the string 'Fizz' given any input, and onther that returns 'Buzz,' given any input. They are saved in a dictionary w ith 3, and 5 as their keys.

```
def three_func(n):
    return 'Fizz'

def five_func(n):
    return 'Buzz'

func_dict = {3:three_func, 5:five_func}

# For all numbers, 0 to 30
for n in range(31):
    # Start with empty string
    toPrint = ''

    # For each key in the "func_dict"
    for k in sorted(list(func_dict.keys())):

        # If number is divisible by the key
        if n % k == 0:

            # Call the associated function
            # Note how the function is called: func_dict[k](n)
            toPrint += func_dict[k](n)

    # if toPrint has not been updated, print the number
    if toPrint == '':
        print(n)
    else:
        print(toPrint)
```

Question 21

```
### GRADED
### Code a function called "apply_functions"

### ACCEPT two lists as inputs:
### The first list will be a list of iterables
### The second list will be some subset of the functions <len>, <sum>, and <type>

### RETURN a list of the same length as the first list, where every element in that list
### is a *tuple* which has had the functions from the second list applied on them.

### e.g. if the lists are [(1,2),[3,4]], and [len, sum, type] The return should be:
### [(2,3,tuple),(2,7,list)].
### Note the tuples are in the order of (length, sum, type); the same order as the passed functions

### YOUR ANSWER BELOW

def apply_functions(list_of_objs, list_of_funcs):
    """
```

```
    Return a list where all the functions in the second argument
    have been applied to the elements of the first argument

    Positional Arguments:
    list_of_objs -- A list of objects (lists and/ or tuples)
    list_of_funcs -- A list of functions that may be applied to the
        objects of the first argument

    Example:
        objs = [(1,2),[1,3,4,5,6,7],[0]]
        funcs = [len,sum]
        print(apply_functions(objs, funcs))
        #--> [(2, 3), (6, 26), (1, 0)]
    """
    toRet = []

    # For all objects in the list of objects
    for i in list_of_objs:
        t = []

        # Apply each of the functions and add to a list
        for f in list_of_funcs:
            t.append(f(i))

        # Turn that list into a tuple and add to list to return
        toRet.append(tuple(t))
    return toRet
```