INFO1110 & 9001 Week 5 Tutorial

Functions and Collections

Tuples

You should remember from the previous tutorial how lists work in Python. You may also have seen the word 'tuple' floating around without being properly defined. Tuples are category of collections in Python are virtually identical to lists, except the elements are immutable. That is to say that once set, they cannot be changed.

Tuples are initialised by ending the value allocated to a variable with a comma. Just as with a list, the elements in a tuple are accessed via an index. For example:

```
a = 1,
print(a)
print(a[0])
```

Try to change the value associated with 'a[0]' what error does this throw?

```
a[0] = 2
```

For neatness, and consistency, tuples are delimited by brackets, which helps to distinguish them from lists.

```
a_tuple = (1, 2, 3, 4)
a_list = [1, 2, 3, 4]
print(a_tuple, a_list)
```

Question 1: Slightly Less Useful Lists

Write a Python program that reads from a file and constructs a tuple where each element of the tuple is the next line of the file. Then convert this tuple to a list by looping over its elements.

You should be able to do each of these operations in one to two lines. If you haven't already, rewrite your approach to use more 'Pythonic' loops.

Dictionaries

With lists using braces and tuples using brackets, it follows to reason that there must exist another form of collections that use the curly braces '{'. As it turns out, there is: the dictionary.

Dictionaries are a collection that uses 'key-value' pairs. Rather than having an index set by a number, the index is instead set by a key which is then associated with the value that you are actually trying to store in the collection. If you think of a physical dictionary, you search for a particular word and are given its definition. In Python dictionaries the word is the 'key' and the definition is the 'value' that you are storing.

The size of a dictionary can be grown arbitrarily as new keys are used to add new values to the dictionary. If an element is accessed using the key, the value can be modified or changed.

```
a_dictionary = {}
a['key'] = 'value'
print(a['key'])
a['another_key'] = ('tuple_values!', 'and again!')
print(a)
```

Question 2: Configuration File

You have a file containing a number of configuration options. Comment lines are prefixed with a '#' symbol. Other lines contain the name of a variable followed by its value. If the value is in quotes it is a string, otherwise it is an integer number.

You are to read the configuration file and create a dictionary where the keys are all the names of the configuration options, and the values are the associated values from the configuration file. Some lines may be completely blank and should be ignored. A sample configuration file is given:

```
# Config file
distance_render 'On'
occular_shading 'Off'

# Garbage values
flying_pigs 3
entangled_qubits 3
```

Question 3: Loops and Dictionaries

Given the following dictionary try looping over and printing out each element of the dictionary.

```
a_dictionary = {'q':'Topological entanglement',
'torus':'t','area law': 1,
 1:'area law',
 2:3,
 2.4:'double'}
```

What is being looped over in the dictionary? How does this differ from looping over a tuple or a list?

Functions

As you write increasingly complex and lengthy programs, you may find yourself duplicating certain sections of code quite often. Occasionally you may belatedly notice that you've made a mistake in the code you have just copied half a dozen times and need to fix it.

This code duplication can be avoided by writing functions. Functions are called with some arguments and have a return value. For example the following function 'square' takes some variable 'a' and returns 'a' multiplied by itself.

```
def square(a):
    return a * a
```

This function can then be called.

```
square(2)
4
```

Any Python code can be contained within a function, including loops, switches, and all the other objects that you have seen and used. As any Python object can be passed as an argument, we can also pass collections and functions.

```
def sum_of_squares(a_list):
        square_sum = sum([a * a for a in a_list])
        return square_sum
```

Question 4: Take it to the Mean

Write a function that takes three arguments. Your function should print the mean of these three numbers and return the number that is the furthest from the mean.

Question 5: Mute and Mutability

Functions in Python treat collections differently from other arguments. If a collection is modified within a function call then the change persists. Changes to characters, integers and non-collection types do not persist. Given that tuples are already immutable, this does not apply to them.

Write a function that takes a list argument and increments each element by 1.

Write a separate function that takes a non-list (integer) argument and increments it by one.

```
list_obj = [1,2,3,4,5]
int_obj = 6
```

Find a way to ensure that the values of the objects in both cases have changed after calling the function.

Question 6: List Adder

Write a function that takes two list arguments. If both lists are of the same length then the function returns a new list comprised of each other list added elementwise (element by element).

If the lists are different sizes it should print some warning indicating this.

Here are some basic test cases, you might want to consider writing some more of your own.

```
test_case_a = [[1,2,3,4], [1,2,3,4]]
print(list_add(a[0], a[1]))
test_case_b = [[1,2,3], [1,2,3,4]]
print(list_add(a[0], a[1]))
```

Args

Let’s consider generalising the notion of an argument to a function using a collection.

```
def sum_list(a_list):
        my_sum = 0
        for i in a_list:
                my_sum = my_sum + i
        return my_sum

sum_list([1,2,3,4,5])
```

Mixing brackets is somewhat ugly and difficult to read (hence difficult to debug), luckily Python has a notion of ‘args’ as a general argument that accepts an arbitrary number of inputs and then combines them into a single tuple. that can then be accessed.

```
def sum_args(*args):
    my_sum = 0
        for i in args:
                my_sum = my_sum + i
        return my_sum

sum_args(1,2,3,4,5)
```

An existing list or tuple can be expanded into a set of *args when the function is called using the same * operator.

```
a = [1,2,3,4,5]
a_sum = sum_args(*a)
```

This gives us the flexibility to pass lists or individual arguments to the same function.

Question 7: Take it to the Mean 2

Write a function that has a single *args input and does the following:

- Prints each number that has been entered
- Finds and prints the mean.
- Returns the argument that is the furthest from the mean.

Keyword Arguments and Default Arguments

Sometimes you want certain arguments to have a default value, and can then be overridden if necessary with another value.

```
def powsum(*args, power=2):
        n_sum = 0
        for i in args:
                n_sum = n_sum + (i ** power)
        return n_sum
```

Question 8: Take it to the Mean 3

Modify your code from the previous section to add a keyword argument ‘print’ that is initially set to false. If print is false then the mean is not printed, but the code returns normally. If print is true then the mean is printed.

Question 9: (Extension) Sorter

Write a function that takes a list as a single argument and sorts the elements of the list from smallest to largest. Remember that changes to collections within functions persist outside the function call.

For a simple (but inefficient) approach to sorting, look for the bubble sort algorithm.