



**Program Code: J620-002-4:2020**

**Program Name: FRONT-END SOFTWARE DEVELOPMENT**

**Title : P03: Data Structures I (List, Tuples)**

**Name: Ooi Caaron**

**IC Number: 990701-07-5837**

**Date : 24/6/23**

**Introduction : Learning the list and tuples, so that can handle big amount of data**

**Conclusion : Still need to practice more**

## **Module P03: Data Structures I (List, Tuples)**

- [List](#)
- [Tuples](#)

### **List**

Today, we're going to introduce the **list** data type, and lists are much more powerful than strings, so whereas for a string, all of the elements had to be characters, in a list, the elements can be anything we want. They could be characters. They could be strings. They could be numbers. They could also be other lists.

Like a string, a **list** is a sequence of values. In a string, the values are characters; in a list, they can be any type. The values in list are called **elements** or sometimes **items**.

List elements are written within square brackets [ ]. Square brackets [ ] access data, with the first element at index 0. Many of the operations defined above for strings also apply to lists. So it can be convenient to think of a string simply as a list of characters.

In [15]:

```
colors = ['red', 'blue', 'green']
print(colors[0])
print(colors[1])
print(colors[2])
print("")

print('red' in colors)
print('black' in colors)
print("")

numbers = [5, 4, 3, 2, 1, 'hello', 2.3]    # in a list, we can have a mix of integer, float, string, etc.
print(len(numbers))

print()
list_of_lists = [[1,2,3],[4,5,6],[7,8,9]]
print(list_of_lists)
print(list_of_lists[1][2])                # figure out what do this index numbers represent

# Empty list
c = []
```

```
red
blue
green
```

```
True
False
```

```
7
```

```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
6
```

In [21]:

```
# Assignment with an = on lists does not make a copy.
# Instead, assignment makes the two variables
# point to the one list in memory.
b = a = [1,2,3,5,6]
a.append(0)
a = [4,5]
print(a)
print(b)

# Note: The above is true for all objects in Python, not only lists.
```

```
[4, 5]
[1, 2, 3, 5, 6, 0]
```

In [42]:

```

months = ["January", "February", "March", "April", "May", "June", "July", \
          "August", "September", "October", "November", "December"]
days = [31,28,31,30,31,30,31,31,30,31,30,31]

# a function that inputs a string with the month and returns the number
# of days in that month.

def month_day(month):
    print(days[months.index(month)])

month_day('June')
month_day('July')
month_day('December')

```

30  
31  
31

**Quick Exercise 1** We have a nested list containing the country's name, capital city and population (in millions). How many times bigger is the population of China, India and US relative to Malaysia?

In [33]:

```

# Nested Lists Example, Let's use population (millions).
countries = [['China','Beijing', 1386],
             ['India','Delhi',1339],
             ['United States','Washington DC', 325],
             ['Malaysia','Kuala Lumpur',31]]

# How many times bigger is the population of China, India, and the US relative to Malaysia?
def population():
    for i in range(3):
        if countries[i][2] > countries[3][2]:
            print(f"{countries[i][0]}'s {countries[i][1]} is {countries[i][2] // countries[3][2]} times bigger than Malaysia")

population()

```

China's Beijing is 44 times greater than malaysia  
 India's Delhi is 43 times greater than malaysia  
 United States's Washington DC is 10 times greater than malaysia

## Mutability

In [4]:

```
# A big difference is strings can't mutate - they can't change the
# existing object.
s = 'Hello'
print((s + '!'))
print(s)
print((s[0]))

#String is immutable
#s[0] = "Y"
```

Hello!  
Hello  
H

In [56]:

```
# But Lists are mutable: once an object is mutable, then we
# have to worry about other variables that might
# refer to the same object.
p = ['H','e','l','l','o']
q = p
p[0] = 'Y'
print(q)
```

['Y', 'e', 'l', 'l', 'o']

In [57]:

```
# Assignment with an = on Lists does not make a copy.
# Instead, assignment makes the two variables
# point to the one List in memory.
num = [4,5,6,8,9]
b = num
print (num)
num.append(0)
print(num)
print(b)
```

[4, 5, 6, 8, 9]  
[4, 5, 6, 8, 9, 0]  
[4, 5, 6, 8, 9, 0]

## List Methods

Here's a list of useful methods (no pun intended!) for list objects:

- `list.append(elem)` -- adds a single element to the end of the list. Common error: does not return the new list, just modifies the original.
- `list.insert(index, elem)` -- inserts the element at the given index, shifting elements to the right.
- `list.extend(list2)` adds the elements in `list2` to the end of the list. Using `+` or `+=` on a list is similar to using `extend()`.

- `list.index(elem)` -- searches for the given element from the start of the list and returns its index. Throws a `ValueError` if the element does not appear (use `"in"` to check without a `ValueError`).
- `list.remove(elem)` -- searches for the first instance of the given element and removes it (throws `ValueError` if not present)
- `list.sort()` -- sorts the list in place (does not return it).
- `sorted(list)` -- return sorted list but keeps the original order of the list
- `list.reverse()` -- reverses the list in place (does not return it)
- `list.pop(index)` -- removes and returns the element at the given index. Returns the rightmost element if

In [7]:

```

colors = ['red', 'green', 'blue']
print(colors)
colors.append('purple')
print(colors)
colors.insert(1, 'yellow')
print(colors)

new_list = ['cyan', 'white']
colors.extend(new_list)

print()
print()

print((colors.index('purple')))
colors.remove('white')
print(colors)
print()

#use del
del colors[1]
print()

# use sort
colors.sort()
colors.sort(reverse=True)
print(colors)
print()

print(colors)
colors.reverse()
print()

print(colors)
print()

print((sorted(colors, reverse=True)))

#pop
colors.pop()
print(colors)

```

```

['red', 'green', 'blue']
['red', 'green', 'blue', 'purple']
['red', 'yellow', 'green', 'blue', 'purple']

```

```

4
['red', 'yellow', 'green', 'blue', 'purple', 'cyan']

```

```

['red', 'purple', 'green', 'cyan', 'blue']

```

```

['red', 'purple', 'green', 'cyan', 'blue']

```

```

['blue', 'cyan', 'green', 'purple', 'red']

```

```

['red', 'purple', 'green', 'cyan', 'blue']
['blue', 'cyan', 'green', 'purple']

```

In [71]:

```
colors = ['red', 'yellow', 'green', 'blue', 'purple', 'cyan']
print(colors)

print((sorted(colors)))
```

```
['red', 'yellow', 'green', 'blue', 'purple', 'cyan']
['blue', 'cyan', 'green', 'purple', 'red', 'yellow']
```

In [76]:

*# If you run the following four lines of codes, what are the final contents of list1 and*

```
list1 = [1,2,3,4]
list2 = [1,2,3,4]

print(list1 is list2)

list1 = list1 + [5, 6]
print(list1)
print()

list1.extend([7,8])          # this 'extends' the original list
print(list1)
print()

list2.append([5, 6])         # this adds a list to a list
print(list2)
```

```
False
[1, 2, 3, 4, 5, 6]

[1, 2, 3, 4, 5, 6, 7, 8]

[1, 2, 3, 4, [5, 6]]
```

## List Slices

Slices work on lists just as with strings, and can also be used to change sub-parts of the list.

In [80]:

```
cars = ['Toyota', 'BMW', 'Honda', 'Benz', 'Isuzu', 'Volkswagen', 'Mazda']

print(cars[1:3])
print(cars[1:-1])
print(cars[:3])
```

```
['BMW', 'Honda']
['BMW', 'Honda', 'Benz', 'Isuzu', 'Volkswagen']
['Toyota', 'BMW', 'Honda']
```

## For & In (Or Loops on Lists)

Python's *for* and *in* constructs are extremely useful, and the first use of them we'll see is with lists. The *for* construct -- *for var in list* -- is an easy way to look at each element in a list (or other collection).

In [92]:

```
# Loop through a list using while
```

```
a = [1,2,3,4,7,8,9,10]
i = 0
len_a = len(a)
while i < len_a:
    print((a[i]))
    i += 1
```

```
1
2
3
4
7
8
9
10
```

In [93]:

```
# Syntax of for and in:
```

```
# for each_element in a_list:
    #Iterate through each element
    #and run this code on each element
    #where each_element refers to the item.
```

```
b = [1,2,3,4,7,8,9,10]
for items in b:
    print(items)
```

```
# Note: much shorter code and easy to read!
```

```
1
2
3
4
7
8
9
10
```



In [94]:

```
print((type(colors)))
print(colors)

for color in colors:
    print(color)
    print((color.title()))
    print()
```

```
<class 'list'>
['red', 'yellow', 'green', 'blue', 'purple', 'cyan']
red
Red
```

```
yellow
Yellow
```

```
green
Green
```

```
blue
Blue
```

```
purple
Purple
```

```
cyan
Cyan
```

In [20]:

```
# sum_list: sums up all values in a list
```

```
def sum_list(your_list):
    sum = 0
    for n in your_list:
        sum = sum + n
    return sum
```

```
squares = [1, 4, 9, 16, 25]
print(sum_list(squares))
```

55

In [96]:

```
# A function that checks whether a number appears on a list  
# returns "Found!" or "Not found!"  
  
def find(num, li):  
    status = "Not found!"  
    for i in li:  
        if num == i:  
            status = "Found"  
            break  
  
    return status  
  
print(find(10, list(range(100))))  
print(find(250, list(range(100))))
```

Found  
Not found!

In [98]:

```
#try with longer lists - 10 million numbers  
print(find(9999999, list(range(10000000))))  
print(find(99, list(range(10000000))))  
print(find('ah', list(range(10000000))))
```

Found  
Found  
Not found!

In [99]:

```
# Define a function that gives the union of lists.  
  
a = ['Orange', 'Banana', 'Apple']  
b = ['Banana', 'Orange', 'Durian']  
  
def union_of_lists(seta, setb):  
    return set().union(seta, setb)  
  
union_of_lists(a, b)
```

Out[99]:

{'Apple', 'Banana', 'Durian', 'Orange'}

**Quick Exercise 2** Complete the function `remove_even()` to print a sub-list of an input list after removing all even numbers from it.

In [23]:

```
num = [7, 8, 120, 25, 44, 20, 27]
def remove_even(x):
    ## write your code here ##
    odd = []
    for i in num:
        if (i % 2 != 0):
            odd.append(i)
    return odd

n1 = remove_even(num)
print(n1)      # answer should be [7, 25, 27]
```

[7, 25, 27]

## Enumerate

To find out the index of the item in the loop we can use `list.index(elem)`, but there is another way to do this using `enumerate()`.

In [100]:

```
my_list = ['happy', 'sad', 'angry']

index_counter = 0
for entry in my_list:
    print("Index of %s is: %s" % (entry, index_counter))
    index_counter += 1

# Here we use another built-in function list([iterable])
# which returns a list whose items are the same and
# in the same order as iterable's items).

list(enumerate(my_list))

for i, element in enumerate(my_list):
    print(i, element)
```

Index of happy is: 0  
Index of sad is: 1  
Index of angry is: 2  
0 happy  
1 sad  
2 angry

In [104]:

```
# We can easily change the start count/index with help of enumerate(sequence, start=0)
for index, item in enumerate(my_list, start = 1):
    print(index, my_list)
```

1 ['happy', 'sad', 'angry']  
2 ['happy', 'sad', 'angry']  
3 ['happy', 'sad', 'angry']

## Range

The `range(n)` function yields the numbers 0, 1, ... n-1, and `range(a, b)` returns a, a+1, ... b-1 -- up to but not including the last number.

In [107]:

```
squares = [1,4,9,16,25]

a = list(range(100, 110))
print(a)

b = list(range(5))
print(b)

for i in range(5):
    print(i)

print()
for i in range(len(squares)):
    print((squares[i]))

for square in squares:
    print(square)
```

```
[100, 101, 102, 103, 104, 105, 106, 107, 108, 109]
```

```
[0, 1, 2, 3, 4]
```

```
0
1
2
3
4
```

```
1
4
9
16
25
1
4
9
16
25
```

## Remove

The `remove()` function can be used as a quick way of removing an entry from the list. However, it can only remove one instance of the entry (if there are more present).

In [110]:

```
my_list= ['Red', 'Green', 'White', 'Black', 'Pink', 'Yellow', 'Red']

my_list.remove('Green')
print(my_list)

my_list.remove('Pink')
print(my_list)

my_list.remove('Red')
print(my_list)           # observe carefully. What did you notice?
```

```
['Red', 'White', 'Black', 'Pink', 'Yellow', 'Red']
['Red', 'White', 'Black', 'Yellow', 'Red']
['White', 'Black', 'Yellow', 'Red']
```

It also throws an error if the entry to be removed does not exist in the list.

In [28]:

```
my_list.remove('Brown')
print(my_list)
```

```
-----
-
ValueError                                Traceback (most recent call las
t)
<ipython-input-28-d75a18c35ba3> in <module>
----> 1 my_list.remove('Brown')
      2 print(my_list)
```

**ValueError:** list.remove(x): x not in list

## Lambda (plus filter and map)

Python supports the creation of anonymous functions (i.e. functions that are not bound to a name) at runtime, using a construct called "lambda".

Normal python function:

```
def f (x):
    return x**2

f(8)
```

while for lambda:

```
g = lambda x: x**2
print(g(8))
```

In [5]:

```
f = lambda x, y : x + y  
f(2,3)
```

Out[5]:

5

Lambda functions are mainly used in combination with the functions `filter()` and `map()` to great effect.

In [6]:

```
def myfunc(n):  
    return len(n)  
  
x = map(myfunc, ('apple', 'banana', 'cherry'))  
  
#convert the map into a list, for readability:  
print(list(x))
```

[5, 6, 6]

In [2]:

```
# example 1  
sentence = 'It is raining cats and dogs'  
  
word = sentence.split() # this function splits a text into individual words  
print(word)  
for i in word:  
    count=len(i)  
    print(count)  
  
g = map(lambda num: len(num), word) # maps each word in a list to its word length  
print(list(g))
```

['It', 'is', 'raining', 'cats', 'and', 'dogs']

2

2

7

4

3

4

[2, 2, 7, 4, 3, 4]

In [31]:

```
# example 2
a = [2,3,5,6,7,8]

def power(a):
    return a**2

# filter ()
h=filter(lambda num1: num1%2==0, a)    # filters each value in the list based on a speci
print(h)
print(list(h))

# map()
h = map(lambda num1: power(num1), a)    # maps each value in the list to the specified fu
print(h)
print(list(h))

<filter object at 0x00000264E51FDCA0>
[2, 6, 8]
<map object at 0x00000264E51FDF10>
[4, 9, 25, 36, 49, 64]
```

## List Comprehensions

A list comprehension is a compact expression to define a whole set. It can be used to construct lists in a very natural and easy way.

The following comprehension says "For each item in numbers, square it and add it to a new list squares."

```
squares = [n*n for n in numbers]
```

While we could also do this using a for-loop, generally,

```
for element in iterable:
    if condition(element):
        output.append(expression(element))
```

it's often more convenient to use list comprehension.

The same general example above gets implemented in a single line by list comprehension:

```
[ expression(element) for element in iterable if condition(element) ]
```

In [32]:

```
# without list comprehension
numbers = [1, 2, 3, 4, 5, 6, 7, 8]

squares = []
for n in numbers:
    squares.append(n*n)
print(squares)

# with list comprehension
squares = [n*n for n in numbers]
print(squares)
```

```
[1, 4, 9, 16, 25, 36, 49, 64]
[1, 4, 9, 16, 25, 36, 49, 64]
```

In [4]:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8]

# find squares of number that lesser or equal to 5
squaresUnderFive = [n*n for n in numbers if n <= 5]

print(squaresUnderFive)
```

```
[1, 4, 9, 16, 25]
```

In [5]:

```
# from the squares found earlier, multiply by 2 for all squares greater than 10
print([s*2 for s in squaresUnderFive if s > 10])
```

```
[32, 50]
```

In [8]:

```
# An example of list comprehension with strings
fruits = ['banana', 'apple', 'cherry', 'lime', 'mango']

my_list = [s.upper() + '!!!' for s in fruits if 'a' in s]

print(my_list)
```

```
['BANANA!!!', 'APPLE!!!', 'MANGO!!!']
```

**Quick Exercise 3** Write one line of Python that takes the list 'a' and makes a new list 'b' that has only the odd elements powered with 3.

In [37]:

```
a = [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

# complete this line
b = [ n**3 for n in a if n % 2 != 0]

print(b) # expected answer: [1, 729, 15625, 117649, 531441]
```

```
[1, 729, 15625, 117649, 531441]
```



## Del

The "del" operator does deletions. In the simplest case, it can remove the definition of a variable, as if that variable had not been defined. Del can also be used on list elements or slices to delete that part of the list and to delete entries from a dictionary.

In [15]:

```
whos
```

Variable	Type	Data/Info
a	list	n=10
b	list	n=5
f	function	<function <lambda> at 0x000002335839D3A0>
fruits	list	n=5
my_list	list	n=3
myfunc	function	<function myfunc at 0x000002335839D040>
x	map	<map object at 0x000002335838ACD0>

In [16]:

```
del myfunc
%whos
```

Variable	Type	Data/Info
a	list	n=10
b	list	n=5
f	function	<function <lambda> at 0x000002335839D3A0>
fruits	list	n=5
my_list	list	n=3
x	map	<map object at 0x000002335838ACD0>

## Tuples

A tuple is a fixed size grouping of elements, such as an (x, y) co-ordinate. Tuples are like lists, except they are immutable and do not change size (tuples are not strictly immutable since one of the contained elements could be mutable like a list).

Tuples are a convenient way of passing around little logical, fixed size bundle of values. Also, a function that needs to return multiple values can just return a tuple of the values.

In [11]:

```
tup = (1, 2.3, 'hi', ['hi'])
print(type(tup))
print((len(tup)))

print((tup[2]))

#tup[2] = 'bye' # tuple element cannot be changed
tup = (1, 2, 'bye')

x,y,z = (42, 13, "hike")

list2 = [1,2,3]

print((z, y))
```

```
<class 'tuple'>
4
hi
('hike', 13)
```

In [17]:

```
def abc(a, b, c):
    d = a+b
    e = c/2
    f = a+b+c
    return d, e, f

tup2 = abc(2, 3, 4)
print(tup2)
```

```
(5, 2.0, 9)
```

Simply "unpack" the tuple by assigning it to individual variables:

In [40]:

```
a, b, c = tup2
print(a)
print(b)
print(c)
```

```
5
2.0
9
```

If you have multiple lists, and intend to take one element from each list (of the same index) and make them into a tuple, use the `zip()` function. Having a list of tuples is sometimes a neat way of accessing each entry collectively.

In [39]:

```
worker = ['Kelly', 'Bill', 'Indran']  
weight = [52.3, 98.7, 70.2]  
t = zip(worker, weight)
```

```
print(t)
```

```
# either way works
```

```
list_t = [j for j in t]
```

```
#list_t = list(t)
```

```
print(list_t)
```

```
<zip object at 0x0000025CC6D308C0>
```

```
[('Kelly', 52.3), ('Bill', 98.7), ('Indran', 70.2)]
```

In [ ]: