COMP 2853

LESSON 5: LISTS AND TUPLES

Agenda – Lesson 5

- Quick Review
  - Lab 4 Review
  - Strings
  - Pre-Reading – Lists and Tuples
- Quiz 4
  - Review Answers
- Assignment 1 – Recommended Structure
- Lists and Tuples
- Lab 5 – We will start it in-class. You will finish it for homework.

Quiz 4

- This is an individual closed book assessment, please do your own work

- You have 20 minutes to complete it

- We will go over the answers afterwards

Course Outline

#	Topics	Quiz	Lab	Assignment
1	Python Setup, Variables, Expressions, Naming Conventions		Lab 1	
2	Logical Operators, Functions and Modules, Main Function	Quiz 1	Lab 2	
3	Branching and Loops	Quiz 2	Lab 3	
4	Strings	Quiz 3	Lab 4	
5	Lists, Tuples, Dictionaries, and Sets	Quiz 4	Lab 5	
6	Pandas	Midterm	Lab 6	Assignment 1 Due

Topics

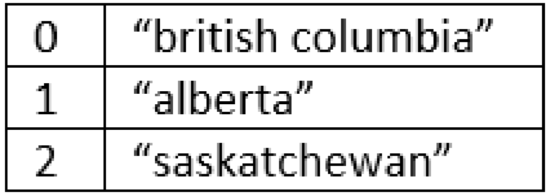
- Containers
- Tuples
- Tuple: multiple return from function
- Lists
- Lists and loops
- List functions and methods
- Lists of lists
- List Comprehensions
- Mutable Operations

Containers

- In python, related data can be grouped together into a single variable.
- For example, instead of declaring one string variable to store the name of a single province, we could declare a container variable to store all of the names of all Canadian provinces.
- There are four container types in python:
- - tuple
- - list
- - set
- - dictionary.
- This lesson examines the first two: tuples and lists.

Tuples

- A tuple contains an ordered sequence of elements (e.g. a bunch of strings for the names of the Canadian provinces).
- Duplicate elements are allowed.
- Elements are automatically given an index, beginning from zero.
- Tuple elements cannot be added, removed, or updated once declared (“immutable”).
- Tuples use (parentheses).
- Example:



0	“british columbia”
1	“alberta”
2	“saskatchewan”

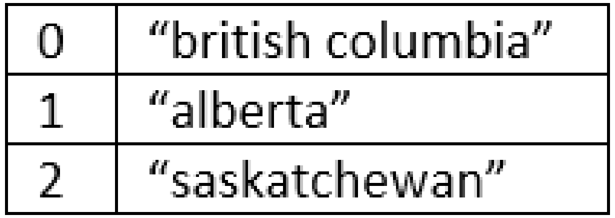
- provinces = ()# empty tuple; very rare to do this
- canadian_provinces = (“british columbia”, “alberta”, “saskatchewan”)

Tuples: Functions and Methods

```
canadian_provinces = (“british columbia”, “alberta”, “saskatchewan”)
```

[Square brackets] give access to individual elements in tuples.

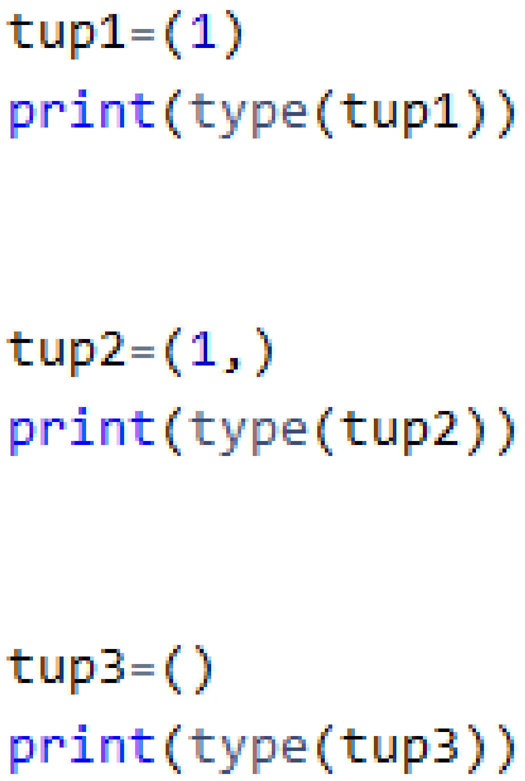
```
print(canadian_provinces[1])          # alberta

print(canadian_provinces[3])           # Exception: IndexError

print(len(canadian_provinces))         # 3
```



0	“british columbia”
1	“alberta”
2	“saskatchewan”

```
car_makers = (“dodge”, “ford”, “honda”, “toyota”, “dodge”)

print(car_makers.count(“dodge”))      # 2; there are 2 instances of “dodge”

print(car_makers.count(“ford”))       # 1

print(car_makers.count(“lamborghini”)) # 0
```

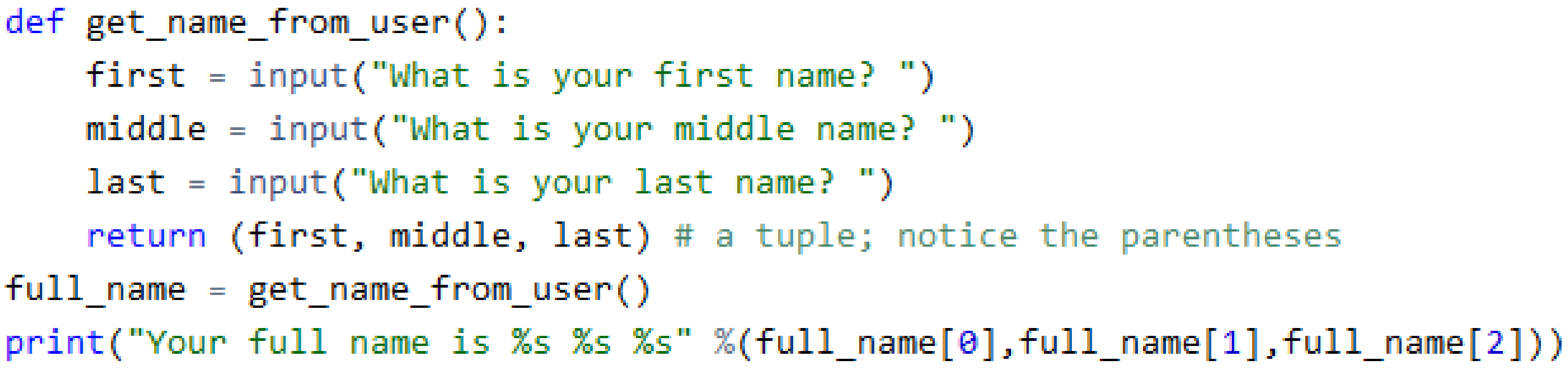
Practice



```
tup1=(1)
print(type(tup1))


tup2=(1,)
print(type(tup2))


tup3=()
print(type(tup3))
```

Tuple: Multiple Return

- Functions only permit one return value, but with a tuple we can return several at once.



```
def get_name_from_user():
    first = input("What is your first name? ")
    middle = input("What is your middle name? ")
    last = input("What is your last name? ")
    return (first, middle, last) # a tuple; notice the parentheses
full_name = get_name_from_user()
print("Your full name is %s %s %s" %(full_name[0],full_name[1],full_name[2]))
```

- OUTPUT:

```
What is your first name? tiger
What is your middle name? tont
What is your last name? woods
Your full name is tiger tont woods
```

Tuples - Alternate

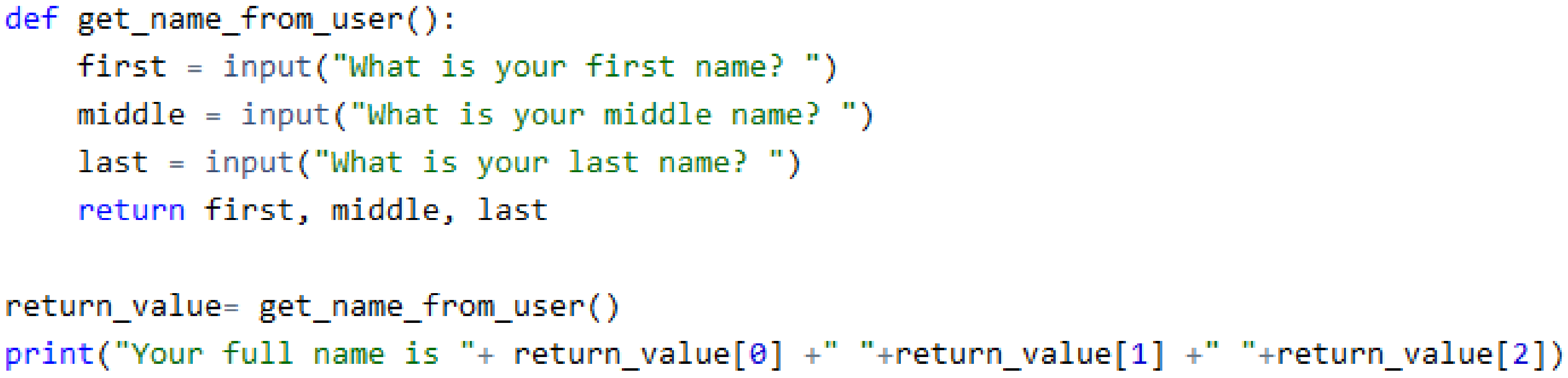
- It is okay to specify a tuple without the () as well:

```
def get_name_from_user():
    first = input("What is your first name? ")
    middle = input("What is your middle name? ")
    last = input("What is your last name? ")
    return first, middle, last  # return <class ‘tuple’>
```

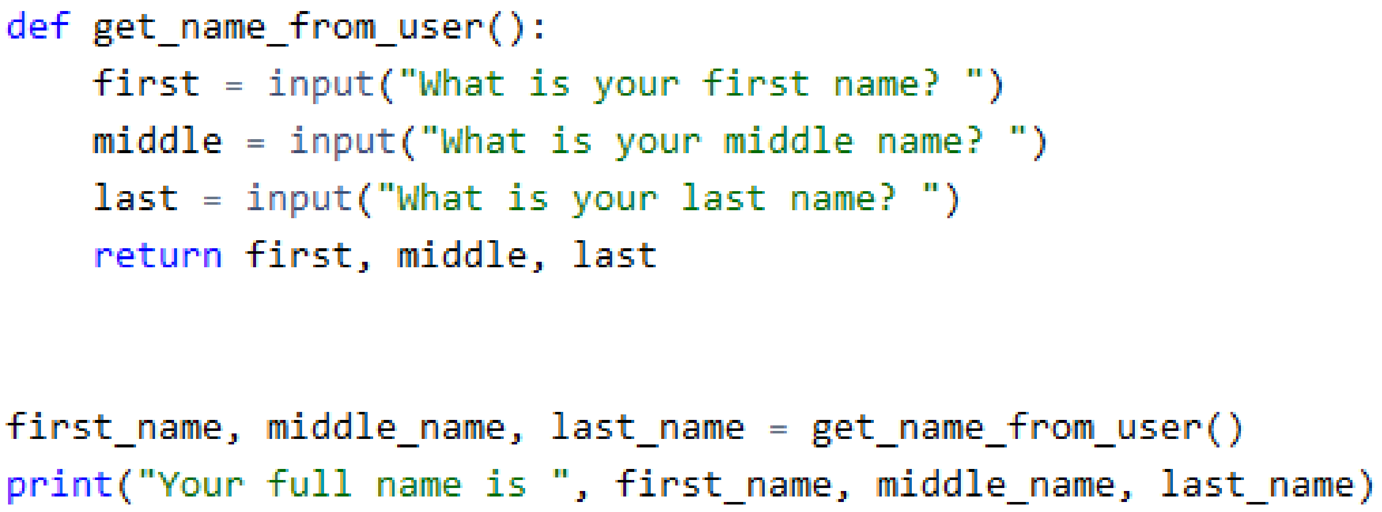
- You can also assign the returned tuple to individual variables:

```
first_name, middle_name, last_name = get_name_from_user()
print("Your full name is %s %s %s" %(first_name, middle_name, last_name)
```

Rewrite code:



```python
def get_name_from_user():
    first = input("What is your first name? ")
    middle = input("What is your middle name? ")
    last = input("What is your last name? ")
    return first, middle, last

return_value= get_name_from_user()
print("Your full name is "+ return_value[0] +" "+return_value[1] +" "+return_value[2])
```



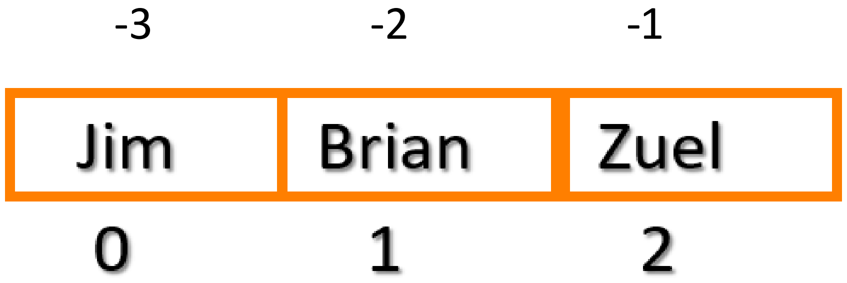
```python
def get_name_from_user():
    first = input("What is your first name? ")
    middle = input("What is your middle name? ")
    last = input("What is your last name? ")
    return first, middle, last


first_name, middle_name, last_name = get_name_from_user()
print("Your full name is ", first_name, middle_name, last_name)
```

List (Array)

- A List is a kind of collection (data structure):
  - A collection allows us to put many values in a single “variable”.
  - A collection is nice because we can carry all multiple values around in one convenient package.
  - my_Friends=[‘Jim’, ‘Brian’, ‘Anu’, ‘Alex’]
  - carryon =[‘socks’, ‘shirt’, ‘perfume’]
  - Arrays in Python are also call lists.
  - Arrays store multiple values of the same data type

Looking Inside Lists

- Elements of the array are referenced sequentially with an index.
- Just like strings, we can get at any single element in a list using an index specified in square brackets



-3	-2	-1
Jim	Brian	Zuel
0	1	2

```
>>> my_Friends = [ Jim', ‘Brian', ‘Zuel' ]
>>> print (my_Friends[1])
>>> Brian
```

Array Methods

- Append: List.append(elem)
- Insert: List.insert(index,elem) Note: 2 arguments
- Extend: list.extend(list2) Note: list.extend(“lucy”) vs. list.extend([“lucy”])
- Index: list.index(elem)
- Remove: list.remove(elm)
- Pop: list.pop() Default: last one or based on index
- Sort: list.sort()
- Reverse: list.reverse()

```
>>> list1.extend('lucy')
>>> list1
['Hello', 'the world', 'l', 'u', 'c', 'y']
>>> list1.extend(['lucy'])
>>> list1
['Hello', 'the world', 'l', 'u', 'c', 'y', 'lucy']
```

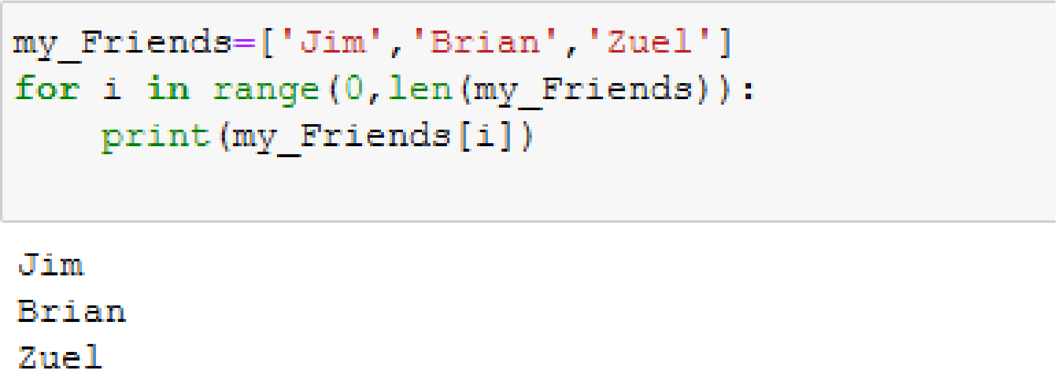
Lists are Mutable

- Strings are "immutable" - we *cannot* change the contents of a string - we must make a new string to make any change
- Lists are "mutable" - we *can* change an element of a list using the index operator

```
>>> fruit="Apple“
>>> fruit[0]
'A'
>>> fruit[0]='B'
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    fruit[0]='B'
TypeError: 'str' object does not support item assignment
>>> new_list=[2,34,56,23,11]
>>> new_list
[2, 34, 56, 23, 11]
>>> new_list[3]=99
>>> new_list
[2, 34, 56, 99, 11]
```

How long is a List

- The len() function takes a list as a parameter and returns the number of *elements* in the list
- Actually len() tells us the number of elements of *any* set or sequence



```
my_Friends=['Jim','Brian','Zuel']
for i in range(0,len(my_Friends)):
    print(my_Friends[i])
```

```
Jim
Brian
Zuel
```

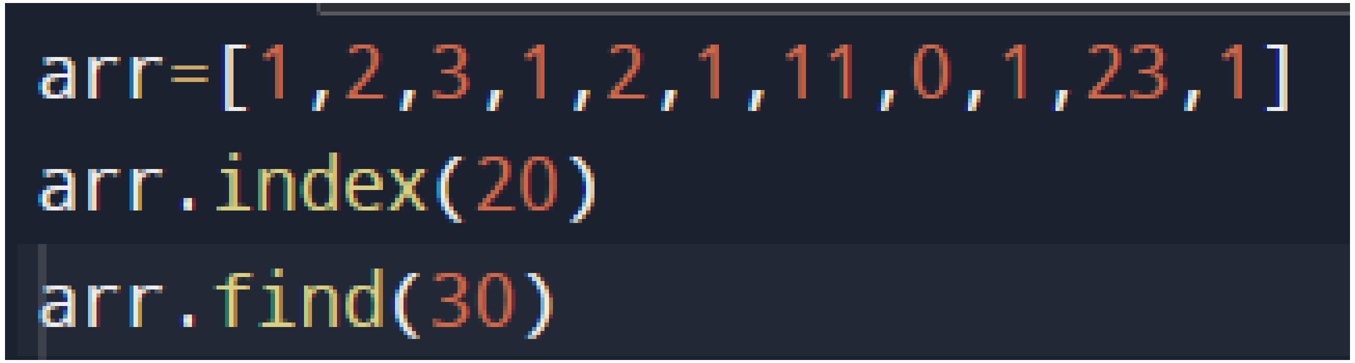
```
>>> greet = 'Hello Bob’
>>> print (len(greet))
9
>>> x = [ 1, 2, 'joe', 99]
>>> print (len(x))
4
```

Adding elements (append() vs. extend())

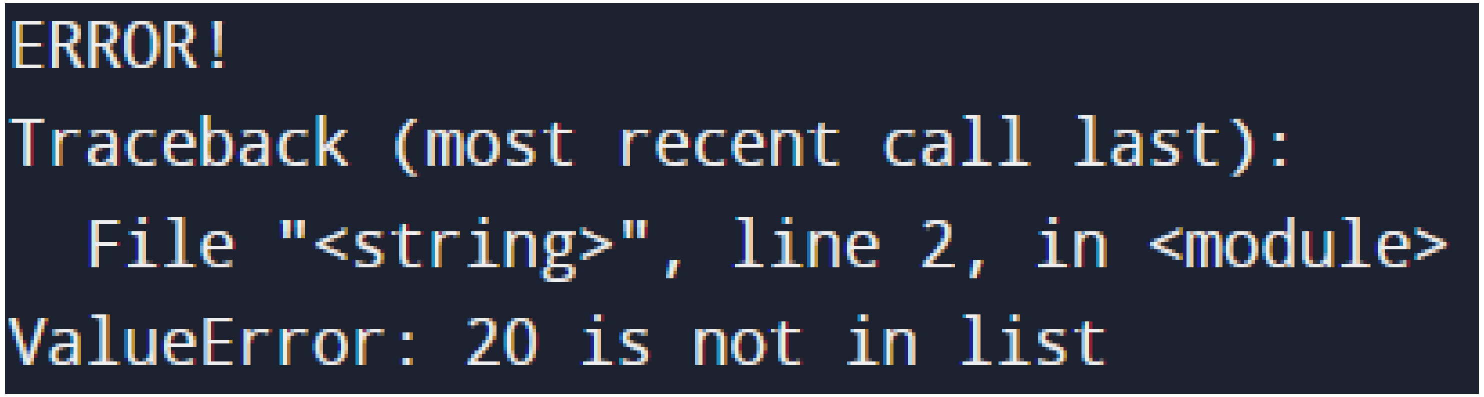
- We can create an empty list and then add elements using the append method
- The list stays in order and new elements are added at the end of the list

```
>>> stuff = list()
>>> stuff.append('book')
>>> stuff.append(99)
>>> print （stuff）
['book', 99]
>>> print (stuff)
['book', 99]
>>> stuff.append(['cookie','22'])
>>> print(stuff)
['book', 99, ['cookie', '22’]]
>>> stuff.extend(['cookie','22'])
>>> stuff
['book', 99, ['cookie', '22'], 'cookie', '22']
```

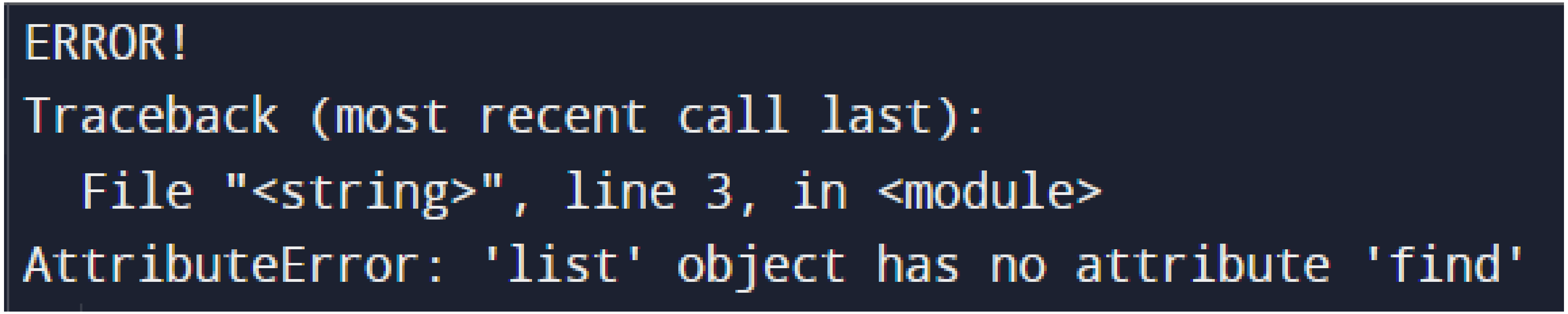
index() vs. find()



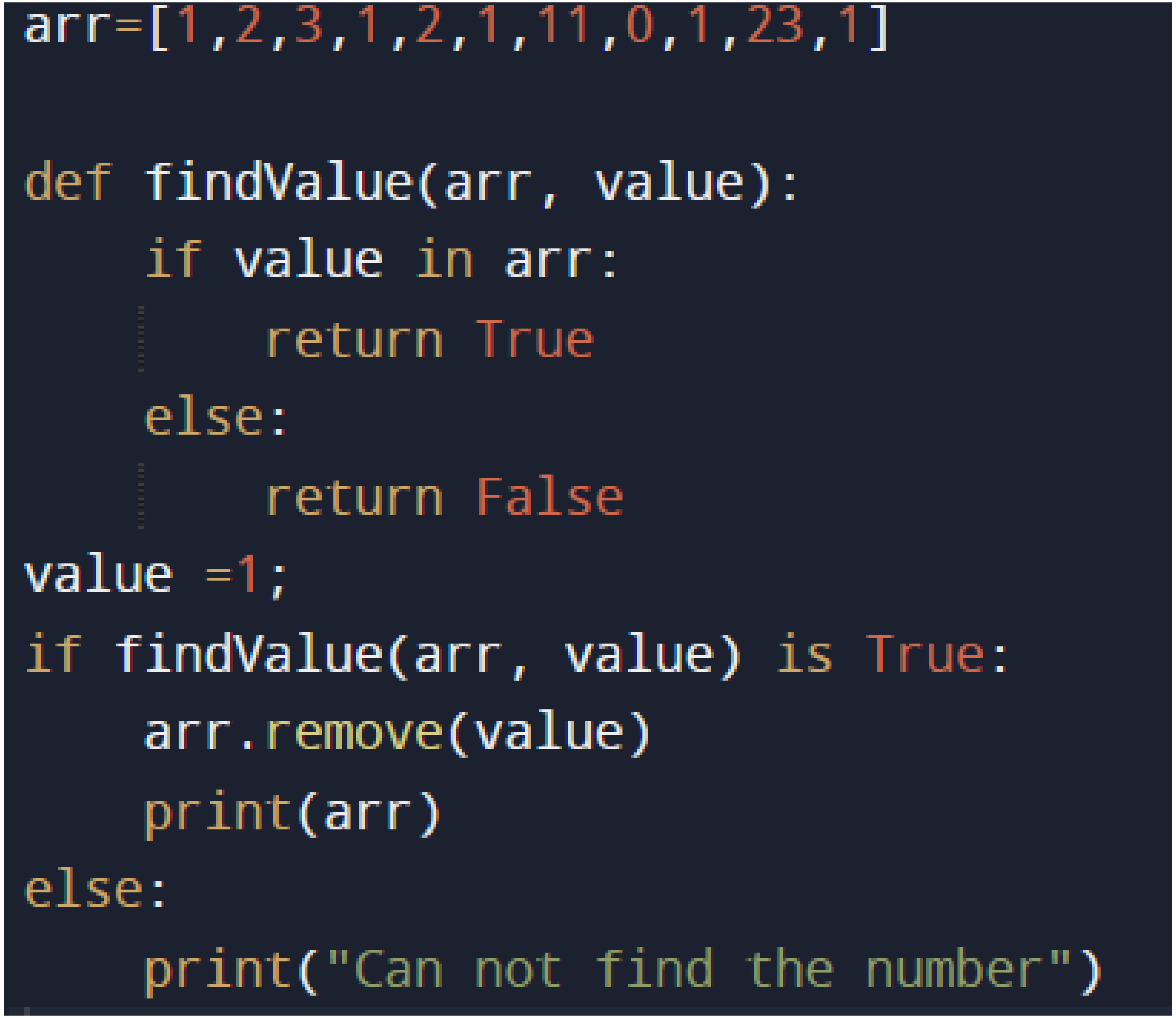
```
arr=[1,2,3,1,2,1,11,0,1,23,1]
arr.index(20)
arr.find(30)
```



```
ERROR!
Traceback (most recent call last):
  File "<string>", line 2, in <module>
ValueError: 20 is not in list
```



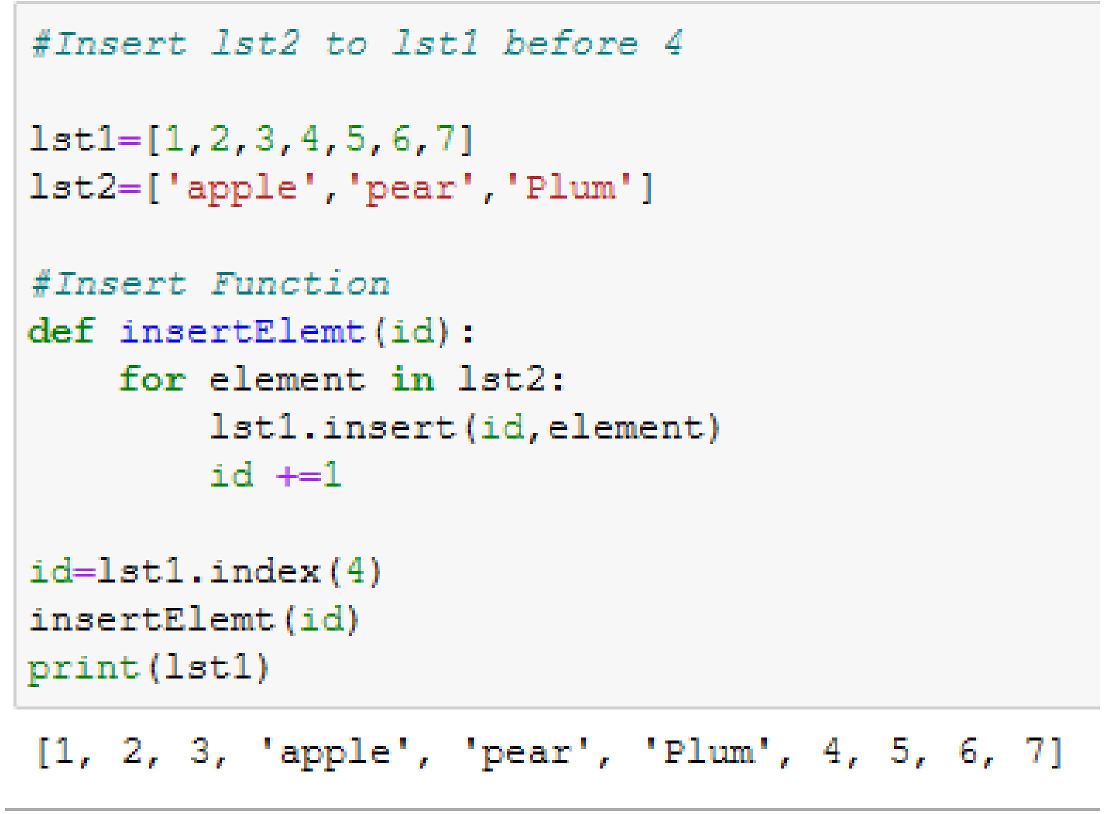
```
ERROR!
Traceback (most recent call last):
  File "<string>", line 3, in <module>
AttributeError: 'list' object has no attribute 'find'
```



```
arr=[1,2,3,1,2,1,11,0,1,23,1]

def findValue(arr, value):
    if value in arr:
        return True
    else:
        return False
value =1;
if findValue(arr, value) is True:
    arr.remove(value)
    print(arr)
else:
    print("Can not find the number")
```

Insert a whole List into another list

```
#Insert lst2 to lst1 before 4
lst1=[1,2,3,4,5,6,7]
lst2=['apple','pear','Plum']

#Insert Function
def insertElemt(id):
    for element in lst2:
        lst1.insert(id,element)
        id +=1
id=lst1.index(4)
insertElemt(id)
print(lst1)
```



```
#Insert lst2 to lst1 before 4

lst1=[1,2,3,4,5,6,7]
lst2=['apple','pear','Plum']

#Insert Function
def insertElemt(id):
    for element in lst2:
        lst1.insert(id,element)
        id +=1

id=lst1.index(4)
insertElemt(id)
print(lst1)
```

```
[1, 2, 3, 'apple', 'pear', 'Plum', 4, 5, 6, 7]
```

pop() vs. remove() vs. del

```
>>> stuff
['book', 99, ['cookie', '22'], 'cookie', '22']
>>> stuff.pop()
'22'
>>> stuff
['book', 99, ['cookie', '22'], 'cookie']
>>> stuff.pop(1)
99
>>> stuff
['book', ['cookie', '22'], 'cookie']
```

```
>>> stuff
['book', 99, ['cookie', '22'], 'cookie', '22']
>>> stuff.remove('22')
>>> stuff
['book', 99, ['cookie', '22'], 'cookie']
>>> stuff.remove(99)
>>> stuff
['book', ['cookie', '22'], 'cookie']
```

```
>>> stuff=['book', 99, ['cookie', '22'], 'cookie', '22']
>>> del stuff[4]
>>> stuff
['book', 99, ['cookie', '22'], 'cookie']
>>> del stuff[1]
>>> stuff
['book', ['cookie', '22'], 'cookie']
```

Lists and Loops

```
index = 0

while index < len(all_letters):

    print("letter is %s\n" % all_letters[index])

    index += 1
```

OUTPUT:

```
letter is a

letter is b

letter is c

letter is d

letter is e

letter is f

letter is g

letter is h

letter is I

letter is j

letter is k
```

List Functions and Methods

- countries = [“japan”, “canada”, “china”, “mexico”, “england”, “greenland”]
- print(**min**(countries)) # canada
- print(**max**(countries)) # mexico
- print(countries.**index**("england")) # 4
- print(countries.**count**("iceland")) # 0 since it appears 0 times
- print(countries.count("canada")) # 1 since it appears 1 time

min – Returns the minimum number in lists of integers or floats
Max – Returns the maximum number in lists of integers or float

Lists of Lists

```
row_0 = ["a", "b", "c", "d", "e", "f", "g", "h"]
row_1 = ["i", "j", "k", "l", "m", "n", "o", "p"]
row_2 = ["q", "r", "s", "t", "u", "v", "w", "x"]
row_3 = ["y", "z"]

table = [row_0, row_1, row_2, row_3]

number_of_rows = len(table)

ii = 0
while ii < number_of_rows:
    print("Row %d" % (ii), end=':: ')
    number_of_letters_in_this_row = len(table[ii])
    jj = 0
    while jj < number_of_letters_in_this_row:
        print((table[ii][jj]), end=' ')
        jj += 1
    print()
    ii += 1
```

OUTPUT

```
Row 0:: a b c d e f g h
Row 1:: i j k l m n o p
Row 2:: q r s t u v w x
Row 3:: y z
```

List Comprehensions

- A list comprehension is a shorter way to create a new list, from another list, based upon conditions that you set.

```
provinces = ["british columbia", "alberta", "saskatchewan", "manitoba", "ontario", "quebec"]

provinces_with_e = [province for province in provinces if "e" in province]

print(provinces_with_e) # ['alberta', 'saskatchewan', 'quebec’]
```

- Here, each element of provinces takes a turn becoming the new variable called “province”. If that element contains the letter e, it is appended to the new list provinces_with_e.

```
newlist = [expression for item in iterable if condition == True]
```

List: Mutable Operations

```
cities = [“Vancouver”, “Surrey”, “Burnaby”,  “Nanaimo”, “Victoria”,
“Port Hardy”]

island_cities = cities[3:] # Slicing

print(island_cities) # Nanaimo, Victoria, Port Hardy

del cities[1] # surrey is deleted

print(cities) # Vancouver, Burnaby, Nanaimo, Victoria, Port Hardy
```

List: Mutable Operations

```
cities = [“Vancouver”, “Surrey”, “Burnaby”, “Nanaimo”, “Victoria”, “Port Hardy”]

cities.extend([“Tsawwassen”, “Coquitlam”]) # adds these to the end of cities

cities.insert(1, “Delta”) # puts delta before surrey and after Vancouver

removed_city = cities.pop(0) # Vancouver is removed and stored in removed_city

cities.sort()

print(cities)

cities.reverse()

print(cities)
```

List Comprehension

- List Comprehension - iterates over a list, modifies each element, and returns a new list consisting of the modified elements.
- Structure:
  - new_list = [*expression* for *name* in *iterable*]
  - new_list = [*expression* for *name* in *iterable* if *condition*]
- A list comprehension has three components:
  - An *expression component* to evaluate for each element in the iterable object.
  - A *loop variable component* to bind to the current iteration element.
  - An *iterable object component* to iterate over (list, string, tuple, enumerate, etc).
  - An optional if statement with *condition* can be added to include or exclude elements.

List Comprehension - Examples

- The benefit of List Comprehension is less code (i.e., code reduction) than a standard for loop

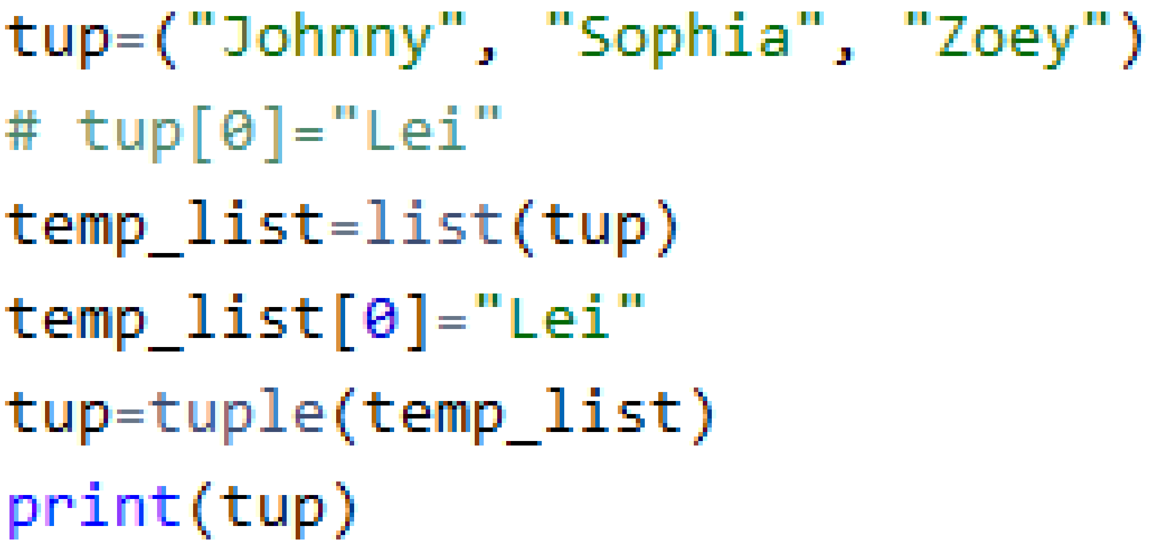
For Loop	Equivalent List Comprehension

my_list = [5, 20, 50]
for i in range(len(my_list)):
    my_list[i] = str(my_list[i])
print(my_list)
 	
['5', '20', '50']
	my_list = [5, 20, 50]
	my_list = [str(i) for i in my_list]
	print(my_list)
	
	['5', '20', '50']

my_list = [[5, 10, 15], [2, 3, 16], [100]]
sum_list = []
for row in my_list:
    if sum(row) > 25:
        sum_list.append(sum(row))
print(sum_list)
 	
[30, 100]
	my_list = [[5, 10, 15], [2, 3, 16], [100]]
	sum_list = [sum(row) for row in my_list if sum(row) > 25]
	print(sum_list)
	
	[30, 100]

How do you modify one element of a tuple in Python?

- **Once a tuple is created, you cannot change its values**. Tuples are unchangeable, or immutable as it also is called.



```
tup=("Johnny", "Sophia", "Zoey")
# tup[0]="Lei"
temp_list=list(tup)
temp_list[0]="Lei"
tup=tuple(temp_list)
print(tup)
```

Sets

- A set can be created by using either the built-in set() function or curly braces {}.

```
my_set = set(‘12334445’)
print(my_set)

The resulting output is:
{‘3’, ‘2’, ‘5’, ‘1’, ‘4’}
```

```
my_set = set(‘repetition’)
print(my_set)

The resulting output is:
{‘e’, ‘r’, ‘p’, ‘o’, ’t’, ’n’, ‘i’}
```

- Strings are treated as separate items unless given as a list or tuple

```
your_set = set([‘foo’, ‘foo’, ‘bar’, ‘car’, ‘star’, ‘car’])
print(your_set)

The resulting output is:
{‘foo’, ‘bar’, ‘star’, ‘car’}
```

Sets

A set can also be created using curly brackets {}

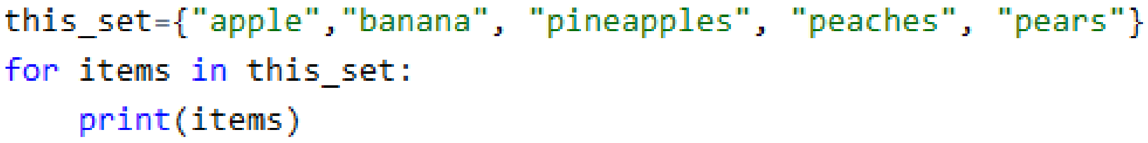
```
my_set = {“hello”, 1, 2, 4}
print(my_set)
```

The resulting output is:

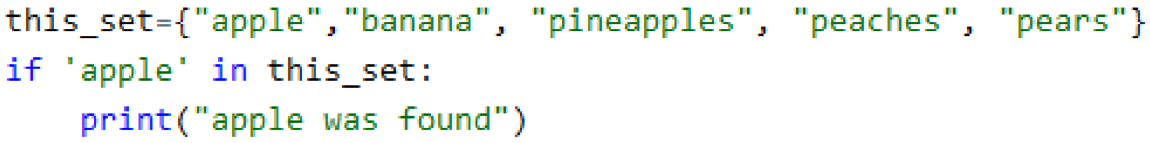
```
{1, 2, 4, “hello”}
```

Sets

- Cannot access items by referring to an index
- Loop through the set
- Use the in keyword

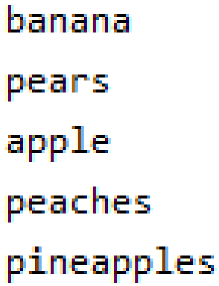


```
this_set={"apple","banana", "pineapples", "peaches", "pears"}
for items in this_set:
    print(items)
```



```
this_set={"apple","banana", "pineapples", "peaches", "pears"}
if 'apple' in this_set:
    print("apple was found")
```

Example output:



```
banana
pears
apple
peaches
pineapples
```

Example output:
apple was found

Sets

- Once a set is created, you cannot change its items, but you can add/remove items
- Add one item using add() method
- Add multiple items using the update() method

```
this_set = {“apple”, “banana”, “cherry”}
this_set.add(“orange”)
this_set.update([“orange”, “mango”, “grapes”])
```

Sets

- Get the number of items in a set using len() method

```
this_set = {“apple”, “banana”, “cherry”}
print(len(this_set))
```

- Remove an item using the remove() method

```
this_set = {“apple”, “banana”, “cherry”}
this_set.remove(“banana”)
```

*if the item does not exist, remove() will raise an error

- Remove an item using the discard() method

```
this_set = {“apple”, “banana”, “cherry”}
this_set.discard(“banana”)
```

*if the item does not exist, discard() will NOT raise an error

Sets

- The clear() method empties the set

```
this_set = {“apple”, “banana”, “cherry”}
this_set.clear()
```

- The del keyword will delete the set completely

```
del this_set
```

Sets

- The union() method returns a new set with all items from each of the sets

```
this_set = {“apple”, “banana”, “cherry”}
that_set = {1, 2, 3}

new_set = this_set.union(that_set)


The resulting new_set:
{1, 2, ‘apple’, 3, ‘cherry’, ‘banana’}
```

- Can also be done using update()

Dictionaries

- A dictionary is an object that stores an unordered collection of data
- Each element has two parts: a key and a value
- **A list element has an index**
  A dictionary value has a key
- A key is like a meaningful index
- Each key is associated with one value, much like each word in the English Language dictionary is associated with a definition
- Elements are commonly referred to as key-value pairs, and sometimes as an item
- To retrieve a specific value from a dictionary, you use the key that is associated with that value

Dictionaries

- You can create a dictionary using {}

```
this_dict = {“brand”: “Ford”, “model”: “Mustang”, “year”: 1964}
print(this_dict)
```

The resulting output:

```
{‘brand’: ‘Ford’, ‘model’: ’Mustang’, ‘year’: 1964}
```

- The keys ‘brand’, ‘model’, and ‘year’ will have the data ‘Ford’, ‘Mustang’, and 1964 associated with them respectively

- **dictionary_name = { “key_name” : “value”, “key_2_name” : “value 2”}**

Containers

- Tuple (value1, value2)
- List [value3, value4]
- Set { value5, value6 }
- Dictionary { key7:value7, key8:value8 }

Let’s build a sample program that uses all four.

Dictionaries

- You can also use the built-in function dict() using keyword arguments to specify the **key-value pairs**

```
this_dict = dict(brand= “Ford”, model= “Mustang”, year= 1964}
print(this_dict)
```

The resulting output:

```
{‘brand’: ‘Ford’, ‘model’: ’Mustang’, ‘year’: 1964}
```

- Or by specifying a list of tuple-pairs

```
this_dict = dict([(‘brand’, ‘Ford’), (‘model’, “Mustang”), (‘year’, 1964)]}
print(this_dict)
```

The resulting output:

```
{‘brand’: ‘Ford’, ‘model’: ’Mustang’, ‘year’: 1964}
```

Dictionaries

- Access values by using square brackets, this_dict[key]
- model_name = this_dict[‘model’]
- Also done by using get() method
- model_name = this_dict.get(‘model’)

model_name = this_dict.get(‘model’, ‘model not found’)

- Square brackets are also used for adding or modifying a value
- this_dict[‘model’] = ‘Ford Focus’
- If the key exists, the value is modified (NO DUPLICATE KEYS ARE POSSIBLE)
- If the key does not exist, the key-value pair is added

Dictionaries

- Delete a key using the del keyword
- del this_dict[‘model’]
- Remove all items from a dictionary using the clear() method
- this_dict.clear()
- Merge dictionaries together using the update() method
- Existing entries are overwritten if the same key exists in the second dictionary (that_dict)
- this_dict.update(that_dict)

Dictionaries

- Remove and return the key value from the dictionary using the pop() method
- If a value does not exist, the default is returned
- model_name = this_dict.pop(‘model’, ‘model not found’)

- Use the in keyword to test for existence of a key in this_dict
- if ‘model’ in this_dict

- len() is used to return the amount of elements in a dictionary
- len(this_dict)

Dictionaries - Iterating

- A for loop can be used to iterate over a dictionary object, the loop variable being set to a key for each iteration
- Print all the keys in the dictionary, one by one

```
this_dict = {‘brand’: ‘Ford’, ‘model’: ‘Mustang’, ‘year’: 1964}
for key in this_dict:
    print(key)
```

The resulting output:

```
brand
model
year
```

Dictionaries - Iterating

- Print all the values in a dictionary using [] notation

```
this_dict = {‘brand’: ‘Ford’, ‘model’: ‘Mustang’, ‘year’: 1964}
for key in this_dict:
    print(this_dict[key])
```

- Can also use the values() method to return the values in a dictionary

```
this_dict = {‘brand’: ‘Ford’, ‘model’: ‘Mustang’, ‘year’: 1964}
for val in this_dict.values():
    print(val)
```

- Output for both

```
Ford
Mustang
1964
```

Lab 5

- There is only one part for this lab.

- We will start the lab in class. You will need to finish it for homework and submit it to the Dropbox on the Learning Hub (Activities -> Assignments -> Lab 5). Due before next class.