

In this session, we are going to explore other container objects

In this session, we are going to learn the following key topics:

- Tuple
- Set
- Dictionary
- Function
- Lambda
- Map
- Reduce
- Filter

## ▼ Tuples

In Python, tuples are similar to lists but they are immutable i.e. they cannot be changed. You would not want to change, such as days of week or dates on a calendar.

In this section, we will get a brief overview of the following key topics:

- 1.) Constructing Tuples
- 2.) Basic Tuple Methods
- 3.) Immutability
- 4.) When to Use Tuples


You'll have an intuition of how to use tuples based on what you've learned about lists. But, the main difference is tuples are immutable.

## Constructing Tuples

The construction of tuples uses () with elements separated by commas where in the arguments will


```
# Can create a tuple with mixed types
t = (1,2,3,8.6,'a')
```

```
# Check len just like a list
type(t)
```


 tuple

```
# Use indexing just like we did in lists
t = ('one',2, 'three')
```

```
t[2]
```

 'three'

```
# Slicing just like a list
t[-1]
```

 'three'

## ▼ Basic Tuple Methods

Tuples have built-in methods, but not as many as lists do. Let's see two samples of tuple built-in m

```
# Use .index to enter a value and return the index
t.index(2)
```

 1


```
# Use .count to count the number of times a value appears
t.count('one')
```

 1

## ▼ Immutability

As tuples are immutable, it can't be stressed enough and add more into it. To drive that point home

```
t[0]= 'valueCantChange'
```

 -----  
**TypeError** Traceback (most recent call last)  
 <ipython-input-7-e35a5068316e> in <module>  
 ----> 1 t[0]= 'valueCantChange'

**TypeError:** 'tuple' object does not support item assignment

SEARCH STACK OVERFLOW

Because tuple being immutable they can't grow. Once a tuple is made we can not add to it.

```
t.append(20)
```

```
t.extend([20,40,70])
```

Unsupported Cell Type. Double-Click to inspect/edit the content.

```
my_tuple = 1,2,3,5,[4,7]
```

```
my_tuple = tuple(my_tuple)
```



```
-----
NameError                                Traceback (most recent call last)
<ipython-input-8-50935e5da5ec> in <module>
----> 1 my_tuple = tuple(my_tuple)

NameError: name 'my_tuple' is not defined
```

SEARCH STACK OVERFLOW

```
type(my_tuple)
```

```
my_tuple[4][1] = 5
```

```
my_tuple
```

## ▼ When to use Tuples

You may be wondering, "Why to bother using tuples when they have a few available methods?"

Tuples are not used often as lists in programming but are used when immutability is necessary. When you need to make sure that it does not get changed then tuple becomes your solution. It provides a

You should now be able to create and use tuples in your programming as well as have a complete

## ▼ Sets

Sets are an unordered collection of *unique* elements which can be constructed using the `set()` function.

Let's go ahead and create a set to see how it works.

```
x = set()
```

```
# We add to sets with the add() method
x.add((3,4,5))
```


```
#Show
x
```



```
{0, 1, 3, 5, 6, 7, 9, 100, 3445, 1212212}
```

```
set_1 = set([5, 2, 7, 2, 1, 88])
set_2 = set([5, 2, 7, 2, 1, 88])
```

```
set_2
```



```
1 2 3 4 5 6 7 8 9 10
```

```
x = {100,1, 6, 7,5,9,3445,0,3,5,1212212}
```

Note that the curly brackets do not indicate a dictionary! Using only keys, you can draw analogies to a dictionary. We know that a set has an only unique entry. Now, let us see what happens when we try to add some more elements to a set.

```
# Add a different element
x.add(2)
```

```
#Show
x
```


```
# Try to add the same element
x.add(1)
```

```
#Show
x
```

Notice, how it won't place another 1 there as a set is only concerned with unique elements! However, we can add multiple elements to a set to get the unique elements. For example:

```
# Create a list with repeats
l = [1,1,2,2,3,4,6,1,5,1]
```

```
# Cast as set to get unique values
set(l)
```




```
{1, 2, 3, 4, 5, 6}
```

**\*\* Given this nested tuple, use indexing to grab the word "hello" \*\***

```
t = (1,2,(3,4),(5,(100,200,('hello'))),23,11),1,7)
```

```
t[3][1][2][0:]
```



```
'hello'
```

## ▼ Dictionaries

We have learned about "Sequences" in the previous session. Now, let's switch the gears and learn about dictionaries. In Python, dictionaries are nothing but hash tables in other programming languages.

In this section, we will learn briefly about an introduction to dictionaries and what it consists of:

- 1.) Constructing a Dictionary
- 2.) Accessing objects from a Dictionary
- 3.) Nesting Dictionaries
- 4.) Basic Dictionary Methods

Before we dive deep into this concept, let's understand what are Mappings?

Mappings are a collection of objects that are stored by a "key". Unlike a sequence, mapping store c  
important distinction since mappings won't retain the order since they have objects defined by a k

A Python dictionary consists of a key and then an associated value. That value can be almost any

## Constructing a Dictionary

```
# Make a dictionary with {} and : to signify a key and a value
my_dict = {True:'value1', 'key2':'value2', 'key1':'valuedfvdvg', 'key1':'abc', 'key2':'val'}
my_dict
```

```
{True: 'value1', 'key2': 'val', 'key1': 'abc'}
```

```
# Call values by their key
my_dict['key2']
```

```
'val'
```

Note that dictionaries are very flexible in the data types they can hold. For example:

```
my_dict = {'key1':123, 'key2':[12,23,33], 'key3':['item0', 'item1', 'item2']}
```

```
#Let's call items from the dictionary
my_dict['key2'][2]
```

```
33
```

```
# Can call an index on that value
my_dict['key3'][2]
```

```
'item2'
```

```
#Can then even call methods on that value
my_dict['key3'][0].capitalize()
```

```
'Item0'
```

We can effect the values of a key as well. For instance:

```
my_dict['key1']
```

```

123
# Subtract 123 from the value
my_dict['key1'] = my_dict['key1'] - 123

#Check
my_dict['key1']

```


 0

Note, Python has a built-in method of doing a self subtraction or addition (or multiplication or division statement. For example:

```

# Set the object equal to itself minus 123
my_dict['key1'] -= 123
my_dict['key1']

```


 -123

We can also create keys by assignment. For instance if we started off with an empty dictionary, we

```

# Create a new dictionary
d = {}
type(d)

```

 dict

```

# Create a new key through assignment
d['animal'] = 'dog'
d

```

 {'animal': 'dog'}

```


# Can do this with any object
d['answer'] = 42

```

```

#Show
d

```

 {'animal': 'dog', 'answer': 42}

## ▼ Nesting with Dictionaries

Let's understand how flexible Python is with nesting objects and calling methods on them. let's have a dictionary:


```

# Dictionary nested inside a dictionary nested inside a dictionary
d = {'key1':{'nestkey':{'subnestkey':'value'}}}

```

That's the inception of dictionaries. Now, Let's see how we can grab that value:

```
# Keep calling the keys
d['key1']['nestkey']['subnestkey']
```

 'value'

## ▼ A few Dictionary Methods


There are a few methods we can call on a dictionary. Let's get a quick introduction to a few methods.

```
# Create a typical dictionary
d = {'key1':1, 'key2':2, 'key3':3}
```


```
# Method to return a list of all keys
f=d.keys()
list(f)
```

 ['key1', 'key2', 'key3']

```
# Method to grab all values
v = d.values()
list(v)
```

 [1, 2, 3]


```
# Method to return tuples of all items
i = d.items()
list(i)
```

 [('key1', 1), ('key2', 2), ('key3', 3)]

**\*\* Given this nest dictionary grab the word "hello". Be prepared, this will be annoying/tricky \*\***

```
d = {'k1':[1,2,3,{'tricky':['oh','man','inception',{'target':[1,2,3,'hello']}]}]}
```


```
d['k1'][3]['tricky'][3]['target'][3]
```

 'hello'

## ▼ Dictionary Comprehensions

Just like List Comprehensions, Dictionary Data Types also support their own version of comprehension commonly used as List Comprehensions, but the syntax is:

```
{x:x**2 for x in range(10)}
```

 {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}

```
{x:x**3 for x in range(0,10,2)}
```

```
{0: 0, 2: 8, 4: 64, 6: 216, 8: 512}
```

One of the reasons is the difficulty in structuring the key names that are not based on the values.

## ▼ Functions

### Introduction to Functions

What is a function in Python and how to create a function?

Functions will be one of our main building blocks when we construct larger and larger amount of c

#### **So what is a function?**

A function groups a set of statements together to run the statements more than once. It allows us inputs to the functions.

Functions allow us to reuse the code instead of writing the code again and again. If you recall strir used to find the length of a string. Since checking the length of a sequence is a common task, you this repeatedly at command.

Function is one of the most basic levels of reusing code in Python, and it will also allow us to start

## ▼ def Statements

Now, let us learn how to build a function and what is the syntax in Python.

The syntax for def statements will be in the following form:

```
def name_of_function(arg1,arg2):
    '''
    This is where the function's Document String (doc-string) goes
    '''
    # Do stuff here
    #return desired result
```

We begin with def then a space followed by the name of the function. Try to keep names relevant & good name for a length() function. Also be careful with names, you wouldn't want to call a function [Python](#) (such as len).

Next, comes the number of arguments separated by a comma within a pair of parenthesis which a them and the function definition with a colon.

Here comes the important step to indent to begin the code inside the defined functions properly. A *whitespace* to organize code and lot of other programming languages do not do this.



Next, you'll see the doc-string where you write the basic description of the function. Using iPython these doc-strings by pressing Shift+Tab after a function name. It is not mandatory to include docs practice to put them as this will help the programmers to easily understand the code you write.

After all this, you can begin writing the code you wish to execute.


The best way to learn functions is by going through examples. So let's try to analyze and understand objects and data structures we learned.

### ▼ Example 1: A simple print 'hello' function

```
def say_hello():  
    print('hello')
```

Call the function

```
say_hello()
```


 hello

### ▼ Example 2: A simple greeting function

Let's write a function that greets people with their name.

```
def greeting(name):  
    print('Hello All ! Good Morning {}'.format (name))
```


```
x = greeting('Everyone')
```

 Hello All ! Good Morning Everyone.

**\*\* Create a function that grabs the email website domain from a string in the form: [\\*\\* user@domain.com](#) would return: domain.com**

```
def domainGet(email):  
    return email.split('@')[-1]
```

```
domainGet('avitech@gmail.com')
```

 'gmail.com'

**\*\* Create a basic function that returns True if the word 'python' is contained in the input string.\*\***

```
def findpy(st):  
    return 'python' in st.lower()
```

```
findpy('Is there a Python class here?')
```

 True

## ▼ Using return

Let's see some examples that use a return statement. Return allows a function to "return" a result in whatever manner a user wants.

### Example 3: Addition function

```
def add_num(num1,num2):
    return num1+num2
```

```
add_num(4,5)
```

 9

```
# Can also save as variable due to return
result = add_num(4,5)
result
```


 9

```
print(result)
```

 9

What happens if we input two strings?

```
print(add_num('one',1))
```

 -----  
**TypeError** Traceback (most recent call last)  
 <ipython-input-50-6855a6f494f6> in <module>  
 ----> 1 print(add\_num('one',1))  
  
 <ipython-input-46-4e98b39fa5d2> in add\_num(num1, num2)  
 1 def add\_num(num1,num2):  
 ----> 2 return num1+num2  
  
**TypeError:** can only concatenate str (not "int") to str

SEARCH STACK OVERFLOW


In Python we don't declare variable types, this function could be used to add numbers or sequence adding in checks to make sure a user puts in the correct arguments into a function.

Let's also start using *break*, *continue*, and *pass* statements in our code. We introduced these during


Now, let's see a complete example of creating a function to check if a number is prime (a common We know a number is said to be prime if that number is only divisible by 1 and itself. Let's write out numbers from 1 to N and perform modulo checks.

```
def is_prime(num):
    for n in range(2,num):
        if num % n == 0:
            print('not prime')
            break
    else: # If never mod zero, then prime
        print('prime')
```

```
is_prime(17)
```

 prime

```
#WAP for filtering sat sunday in a month where month starts with sat
month = list(range(1,31))
def weekend_check(month):
    weekend = []
    for i in month:
        if i < 3:
            weekend.append(i)
        elif i % 7 == 0:
            weekend.append(i+1)
            weekend.append(i+2)
    return weekend
weekend_check(month)
```

 [1, 2, 8, 9, 15, 16, 22, 23, 29, 30]

**\*\* Create a function that counts the number of times the word "dog" occurs in a string. Again ignore**

```
def countDog(st):
    count = 0
    for word in st.lower().split():
        if word == 'dog':
            count += 1
    return count
```

```
countDog('This dog runs faster than the other dog dude!')
```

 2

**\*You are driving a little too fast, and a police officer stops you. Write a function to return one of 3 possible "Big Ticket". If your speed is 60 or less, the result is "No Ticket". If speed is between 61 and 80 incl**

81 or more, the result is "Big Ticket". Unless it is your birthday (encoded as a boolean value in the parameter `is_birthday`).

```
def caught_speeding(speed, is_birthday):
```

```
    if is_birthday:
        speeding = speed - 5
    else:
        speeding = speed

    if speeding > 80:
        return 'Big Ticket'
    elif speeding > 60:
        return 'Small Ticket'
    else:
        return 'No Ticket'
```

```
caught_speeding(81, True)
```

```
'Small Ticket'
```

## ▼ Lambda Function

it is called as anonymous functions. do not defined by `def` keyword return expression but not value. keyword these functions does not having any name `lambda arguments: expression`

```
sum = lambda x,y:x+y
a = int(input("enter a: "))
b = int(input("enter b: "))
print("sum=", sum(a,b))
```

```
enter a: 7
enter b: 6
sum= 13
```

#WAP using lambda function to get cube of the number

```
cube = lambda num:num**3
cube(3)
```

```
27
```

## ▼ map()

The `map()` is a function that takes in two arguments:

1. A function
2. A sequence iterable.

In the form: `map(function, sequence)`

The first argument is the name of a function and the second a sequence (e.g. a list). `map()` applies sequence. It returns a new list with the elements changed by the function.

We'll start with two functions:

```
def square(x):  
    return x*x  
temp = [0, 22.5, 40,100]
```

```
def summ(x, y ):  
    return x+y
```

```
temp = list(map(summ,temp,temp))
```

```
temp
```

```
[0, 45.0, 80, 200]
```

Now let's see `map()` in action:

```
temps = list(map(square, temp))
```

```
#Show  
temps
```

```
[0, 506.25, 1600, 10000]
```

In the example above, we haven't used a lambda expression.

```
list(map(lambda x : x*x, temp))
```

```
[0, 506.25, 1600, 10000]
```

Map is more commonly used with lambda expressions since the entire purpose of a `map()` is to se

`map()` can be applied to more than one iterable. The iterables must have the same length.

For instance, if we are working with two lists-`map()` will apply its lambda function to the elements of the first list, then to the elements with the 1st index until the nth index is reached.


For example, let's map a lambda expression to two lists:

```
a = [1,2,3,4]  
b = [5,6,7,8]  
c = [9,10,11,12,4,5]
```

```
list(map(lambda x,y:x+y,a,b))
```

```
def sum(x,y,z):
    return x+y+z
```

```
# Now all three lists
list(map(sum, a,b,c))
```

 [15, 18, 21, 24]

In the above example, the parameter 'x' gets its values from the list 'a', while 'y' gets its values from your own example to make sure that you completely understand mapping more than one iterable.

## ▼ reduce()

The function `reduce(function, sequence)` continually applies the function to the sequence. It then returns the final result. If `seq = [s1, s2, s3, ... , sn]`, calling `reduce(function, sequence)` works like this:

- At first the first two elements of sequence will be applied to function, i.e. `func(s1,s2)`
- The list on which `reduce()` works looks like this: `[ function(s1, s2), s3, ... , sn ]`
- In the next step the function will be applied on the previous result and the third element of the sequence
- The list looks like: `[ function(function(s1, s2),s3), ... , sn ]`
- It continues like this until just one element is left and return this element as the result of `reduce()`

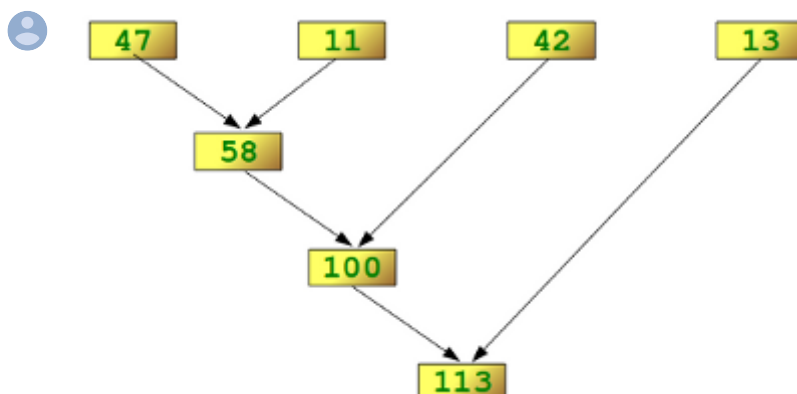
Let's see an example:

```
from functools import reduce
lst = ['sdfs', 'sdfsfsdf', 'sdf']
reduce(lambda a,b: a+b, lst)
```

 'sdfssdfsfsdfsdf'

Let's look at a diagram to get a better understanding of what is going on here:

```
from IPython.display import Image
Image('http://www.python-course.eu/images/reduce_diagram.png')
```



```
#find add using reduce
```

```

#Find max using reduce
l = [47, 11, 42, 13]
reduce(lambda a,b:a+b,l)

```

 113

Note how we keep reducing the sequence until a single final value is obtained. Let's see another example

```

#Find the maximum of a sequence (This already exists as max())
max_find = lambda a,b: a if (a > b) else b
lst =[47,11,42,13]

```

```

#Find max
reduce(max_find,lst)

```

 47

## ▼ filter

The function `filter(function, list)` offers a convenient way to filter out all the elements of an iterable. The function `filter(function(),l)` needs a function as its first argument. The function needs to return a boolean value. The function will be applied to every element of the iterable. Only if the function returns "True" will the element be included in the result.

Let's see some examples:

```

#First let's make a function
def even_check(num):
    if num%2 ==0:
        return True

```


Now let's filter a list of numbers. Note that putting the function into filter without any parenthesis is fine because functions are objects as well.

```

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

list(filter(even_check,lst))


```

 [2, 4, 6, 8]

```

list(filter(lambda x:x%2==0, lst))

```

 [2, 4, 6, 8]

filter() is more commonly used with lambda functions, this because we usually use filter for a quick


```
list(map(lambda x: x%2==0, lst))
```

 [False, True, False, True, False, True, False, True]

```
#WAP for filtering the age of all students above 20
students = [18, 17, 19, 20, 25, 16, 23]
```

```
#WAP for finding the youngest student age from the list
students = [18, 17, 19, 20, 25, 16, 23]
```

```
list(filter(lambda age : age>20, students))
```

 [25, 23]

```
#WAP for filtering the age of all students above 20
students = [18, 17, 19, 20, 25, 16, 23]
reduce(lambda age1, age2 : age1 if (age1<age2) else age2, students)
```


 16

```
# function that filters vowels
def fun(variable):
    letters = ['a', 'e', 'i', 'o', 'u']
    if (variable in letters):
        return True
    else:
        return False
```

```
# sequence
sequence = ['a', 'v', 'i', 's', 'h', 'k', 'a', 'r']
```

```
# using filter function
filtered = filter(fun, sequence)
```

```
print('The filtered letters are:')
for s in filtered:
    print(s)
```


 The filtered letters are:  
a  
i  
a

**\*\* Use lambda expressions and the filter() function to filter out words from a list that start with the**  
['soup','dog','salad','cat','great'] should be filtered down to: ['soup','salad']

```
seq = ['soup', 'dog', 'salad', 'cat', 'great']
```



```
list(filter(lambda word: word[0]=='s',seq))
```

 ['soup', 'salad']