Welcome to session 2 of week 1

Today we will cover

- Introduction to python
- Basic Data types
- Strings
- List, set, Tuples, Dictionaries
- Operators

Python Installation and Basics

Introduction:

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance. Python supports modules and packages, which encourages program modularity and code reuse. The Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed.

Installing Python

Install letest version of Python by visiting - https://www.python.org/

Anaconda Python Distribution

Anaconda is an open-source package manager, environment manager, and distribution of the Python and R programming languages. It is commonly used for large-scale data processing, scientific computing, and predictive analytics, serving data scientists, developers, business analysts, and those working in DevOps.

Anaconda offers a collection of over 720 open-source packages, and is available in both free and paid versions. The Anaconda distribution ships with the conda command-line utility. You can learn more about Anaconda and conda by reading the Anaconda Documentation pages.

Why Anaconda?

- User level install of the version of python you want
- Able to install/update packages completely independent of system libraries or admin privileges
- conda tool installs binary packages, rather than requiring compile resources like pip - again, handy if you have limited privileges for installing necessary libraries.
- More or less eliminates the headaches of trying to figure out which version/release of package X is compatible with which version/release of package Y, both of which are required for the install of package Z
- Comes either in full-meal-deal version, with numpy, scipy, PyQt, spyder IDE, etc. or in minimal / alacarte version (miniconda) where you can install what you want, when you need it
- No risk of messing up required system libraries

Installing on Windows

1. Download the Anaconda installer.
2. Optional: Verify data integrity with MD5 or SHA-256. More info on hashes
3. Double click the installer to launch.

   NOTE: If you encounter any issues during installation, temporarily disable your anti-virus software during install, then re-enable it after the installation concludes. If you have installed for all users, uninstall Anaconda and re-install it for your user only and try again.
4. Click Next.
5. Read the licensing terms and click I Agree.
6. Select an install for “Just Me” unless you’re installing for all users (which requires Windows Administrator privileges).
7. Select a destination folder to install Anaconda and click Next.

   NOTE: Install Anaconda to a directory path that does not contain spaces or unicode characters.

   NOTE: Do not install as Administrator unless admin privileges are required.
8. Choose whether to add Anaconda to your PATH environment variable. We recommend not adding Anaconda to the PATH environment variable, since this can interfere with other software. Instead, use Anaconda software by opening Anaconda Navigator or the Anaconda Command Prompt from the Start Menu.

9. Choose whether to register Anaconda as your default Python 3.6. Unless you plan on installing and running multiple versions of Anaconda, or multiple versions of Python, you should accept the default and leave this box checked.
10. Click Install. You can click Show Details if you want to see all the packages Anaconda is installing.
11. Click Next.
12. After a successful installation you will see the “Thanks for installing Anaconda”
13. You can leave the boxes checked “Learn more about Anaconda Cloud” and “Learn more about Anaconda Support” if you wish to read more about this cloud package management service and Anaconda support. Click Finish.
14. After your install is complete, verify it by opening Anaconda Navigator, a program that is included with Anaconda. From your Windows Start menu, select the shortcut Anaconda Navigator. If Navigator opens, you have successfully installed Anaconda.

Installing on macOS

1. Download the graphical macOS installer for your version of Python.
2. OPTIONAL: Verify data integrity with MD5 or SHA-256. For more information on hashes, see What about cryptographic hash verification?.
3. Double-click the .pkg file.
4. Answer the prompts on the Introduction, Read Me and License screens.
5. On the Destination Select screen, select Install for me only.

   NOTE: If you get the error message “You cannot install Anaconda in this location,” reselect Install for me only.

6. On the Installation Type screen, you may choose to install in another location. The standard install puts Anaconda in your home user directory:
7. Click the Install button.
8. A successful installation.

Installing on Linux

1. In your browser, download the Anaconda installer for Linux.
2. Optional: Verify data integrity with MD5 or SHA-256. (For more information on hashes, see cryptographic hash validation.)

Run the following:

md5sum /path/filename OR:

sha256sum /path/filename

3. Verify results against the proper hash page to make sure the hashes match.
4. Enter the following to install Anaconda for Python 3.6:

   bash ~/Downloads/Anaconda3-4.4.0-Linux-x86_64.sh OR

   Enter the following to install Anaconda for Python 2.7:

   bash ~/Downloads/Anaconda2-4.4.0-Linux-x86_64.sh NOTE: You should include the bash command regardless of whether you are using Bash shell.

   NOTE: Replace actual download path and filename as necessary.

   NOTE: Install Anaconda as a user unless root privileges are required.
5. The installer prompts “In order to continue the installation process, please review the license agreement.” Click Enter to view license terms.
6. Scroll to the bottom of the license terms and enter yes to agree to them.
7. The installer prompts you to click Enter to accept the default install location, press CTRL-C to cancel the installation, or specify an alternate installation directory. If you accept the default install location, the installer displays **'PREFIX=/home/"user"/anaconda"2 or 3"' and continues the installation. It may take a few minutes to complete.**

   The installer prompts “Do you wish the installer to prepend the Anaconda"2 or 3" install location to PATH in your /home/"user"/.bashrc ?” We recommend yes.

   NOTE: If you enter “no”, you will need to manually specify the path to Anaconda when using Anaconda. To manually add the prepended path, edit file .bashrc to add ~/anaconda"2 or 3"/bin to your path manually using:
8. **export PATH="/home/"user"/anaconda"2 or 3"/bin:$PATH"** Replace /home/"user"/anaconda"2 or 3" with the actual path.
9. The installer finishes and displays “Thank you for installing Anaconda"2 or 3"!”
10. Close and open your terminal window for the installation to take effect, or you can enter the command source ~/.bashrc.

    Enter conda list. If the installation was successful, the terminal window should display a list of installed Anaconda packages.

    NOTE: Power8 users: Navigate to your Anaconda directory and run this command:

    conda install libgfortran

Jupyter Notebooks

The notebook is a web application that allows you to combine explanatory text, math equations, code, and visualizations all in one easily sharable document.

Notebooks have quickly become an essential tool when working with data. You'll find them being used for data cleaning and exploration, visualization, machine learning, and big data analysis. Typically you'd be doing this work in a terminal, either the normal Python shell or with IPython. Your visualizations would be in separate windows, any documentation would be in separate documents, along with various scripts for functions and classes. However, with notebooks, all of these are in one place and easily read together.

Notebooks are also rendered automatically on GitHub. It's a great feature that lets you easily share your work. There is also http://nbviewer.jupyter.org/ that renders the notebooks from your GitHub repo or from notebooks stored elsewhere

Installing and Launching Jupyter Notebooks

By far the easiest way to install Jupyter is with Anaconda. Jupyter notebooks automatically come with the distribution. You'll be able to use notebooks from the default environment.

To install Jupyter notebooks in a conda environment, use **conda install jupyter notebook**.

Jupyter notebooks are also available through pip with **pip install jupyter notebook**.

Today we will cover

- *** Introduction to python - current***
- Basic Data types
- Strings
- List, set, Tuples, Dictionaries
- Operators
- Conditional statements

Basic datatypes in python

Basics of Python

We will be discussing all the basics required in Python for this Project. This session covers all the basics of Python Programming language.

We will follow this notebook for this session

This notebooks will go through the basic topics in order as mentioned:

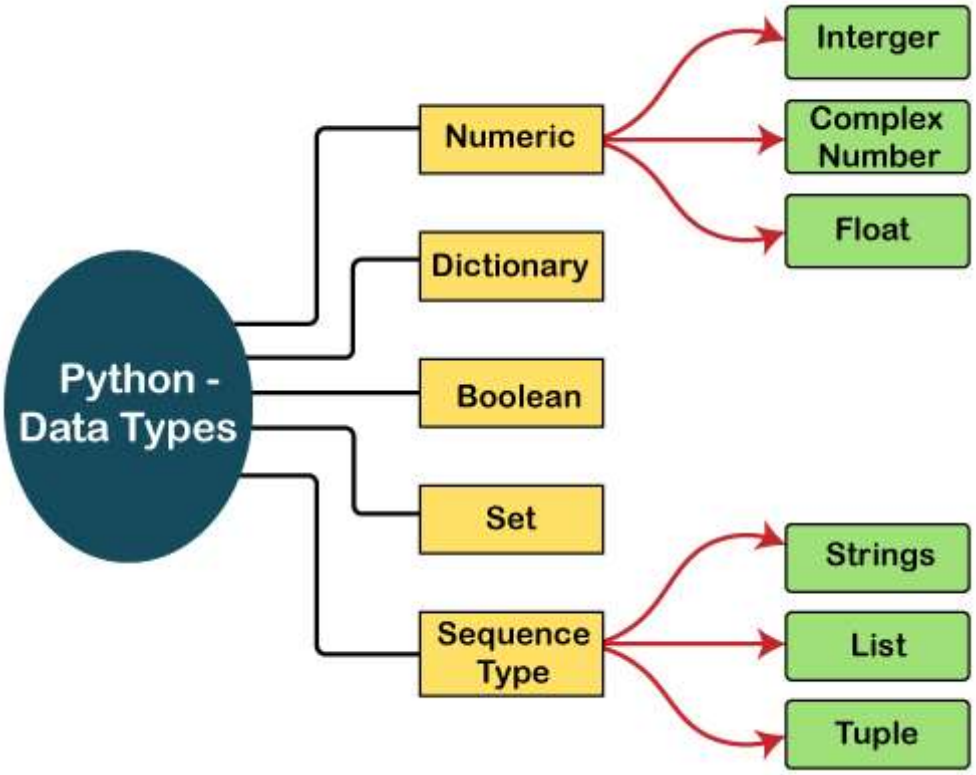
- Input and Output
  - Using interface - input(), print()
  - Using files - open(), read(), write(), close() - Not covered.
- Data types
  - Numeric - int, float, complex
  - Strings - str
  - Booleans - bool
  - Lists - list
  - Dictionaries - dict
  - Tuples - tuple
  - Sets - set
- Operators
  - Arithmetic +, -, , /, //, %, *
  - Bitwise &, |, ^, ~, <<, >> - not covered
  - Assignment =, (arithmetic operator)=, (Bitwise operator)=
  - Logical and, or, not
  - Comparison ==, !=, >, <
  - Identity is, is not
  - Membership in, not in
- Branching
  - if, elif, else Statements
- Looping
  - for Loops
  - while Loops

```
# Know what version of python we are using
from platform import python_version
python_version()
```

```
!python --version
```

Input & Output, Data types and Variable Assignment

- In python based on the value assgined to the variable the data type is automatically assigned and type() funtion can be used to get the data type of any variable



Numeric literals

1. Numeric literals that are without a decimal point are integers
2. Numbers that have decimal points are floats
3. complex numbers are those that have both real and imaginary values and can be created by using the notation x + yj where x is the real component and y is the imaginary component.

```
### Assign an numeric values to the variable, then print it.

my_int=2
my_float=3.5

print('Type of my_int: ', type(my_int) , 'Value in my_int: ', my_int )
print('Type of my_int: ', type(my_float) , 'Value in my_int: ', my_float )
```

Arithmetic Operations

They are used on Numeric data to perform mathematical operations. Possible arithmetic operations in python are: addition +, subtraction -, multiplication , *division /, floor division //,*

*modulo %, exponentiation **

```
a=3
print('{} + 2 = '.format(a), a+2)
print('{} - 2 = '.format(a), a-2)
print('{} * 2 = '.format(a), a*2)
print('{} / 2 = '.format(a), a/2)

# Floor division #  it will find out the integer value smaller than the quotient
print('{} // 2 = '.format(a), a//2)

#Reminder
print('{} % 2 = '.format(a), a%2)

# exponent
print('{} ** 2 = '.format(a), a**3)
```

Strings Literals:

- Strings are sequence of characters.
- String literals can be defined with any of single quotes ('), double quotes (") or triple quotes (''' or """). All give the same result with two important differences.
- If you quote with single quotes, you do not have to escape double quotes and vice-versa.
- There is a special string called docstring which is spans to multiple lines, three single or double quotes are used to assign such text.

```
### Assign an string to the variable, then print it.

my_str1='single quotes'
my_str2="double quotes"

print('Type of my_str1: ', type(my_str1) , 'Value in my_str1: ', my_str1 )
print('Type of my_str2: ', type(my_str2) , 'Value in my_str2: ', my_str2 )


#Docstrings are mostly used for documentation purposes. Like when we want to provide an ov
str_cont = """I'm Sai
            your trainer from spartificial
            our class is asteriod size pred"""

print('Type of str_cont: ', type(str_cont) , '\nValue in my_str1: ', str_cont )

str_cont
```

Strings functions:

There are multiple built-in funtions for string. Detailed list can be found in the notes shared.

In python there is help() method to provide more detail on the provided object

```
help(str)
```

```
string="welcome to spartificial          "

## These methods covert the string but will not replace the original string

print(len(string)) # Provides the total number of characters, count() can be used to get n
print(string.capitalize()) # Changes first char to capital
print(string.upper())   # Changes entire string to uppercase, lower() can be for vice vers
print(string.replace('e','2')) # Replaces all occurences of character with provided charac
print(string.strip()) # rstrip() and lstrip() to strip right and left spaces respectively


# String methods that check the string for a condition and returns boolean value based on

print(string.islower()) # returns True is it's lowercase, isupper() can be used for upper

print(string.isdigit()) # Returns True if string has only numbers in it, isalpha() can be
a = '2353'
print(a.isdigit()) # isalnum() can be used to check if the string is mix of both alphabets

print(string.isspace()) # Returns True if string has only numbers in it, isalpha() can be



# Concatenate strings!
string = 'I am your instructor'
print(string + ' in this research project')

# Reverse a string
print(''.join(reversed(string)))


# Split function gives the list of words in sentence, it splits by spaces by default.
string.split()
```

Strings Operations:

- As mentioned earlier as string is a sequence of characters we and traverse through string using index.
- In python index starts from 0
- Index can be provided in square braces to access a character in specific location
- We can get multiple characters providing staring and ending positions

```
# Get character in Position 0
print('character at 0th positon: ', string[0])

# Get characters in from 5 till 9, note characters from positions 5,6,7,8 are fetched
print('characters from 5 till 9th positions: ', string[5:9])
```

```
# Negative indexing is possible
print('character at last positon: ', string[-1])

# Reversing the string
print('Obtain characters from last to first: ', string[::-1])
```

format()

It is used to pass the arguments and populate place holders in string

```
proj_id=7
string = "'I am your instructor in this research project', and Project ID {}"
print(string.format(proj_id))
```

```
# using .format() for displaying messages
num = 12
name = 'animesh'
print('My number is: {one}, and my name is: {two}'.format(one=num,two=name))
```

```
print('My number is: {}, and my name is: {}'.format(num,name))
```

```
# Printing using f string method

name = 'Spartificial'
age = 1
print(f"Hello, My name is {name} and I'm {age} years old.")
```

Type casting

change the datatype explicitly.

```
my_int=input()
my_float=input()
my_str=input()
print('Type of my_int: ', type(my_int) , 'Value in my_int: ', my_int )
print('Type of my_int: ', type(my_float) , 'Value in my_int: ', my_float )
print('Type of my_int: ', type(my_str) , 'Value in my_int: ', my_str )
```

```
my_int=int(input())
my_float=float(input())
my_str=input()
print('Type of my_int: ', type(my_int) , 'Value in my_int: ', my_int )
print('Type of my_int: ', type(my_float) , 'Value in my_int: ', my_float )
print('Type of my_int: ', type(my_str) , 'Value in my_int: ', my_str )
```

```
# Take a number as input from user (hint: use type casting) and assign it to variable diam
```

```
# Calculate radius and display it
# Calculate area of circle for the obtained radius and display it in single line

diameter=int(input())

radius = diameter / 2
print('Radius of circle: ', radius)

area= 3.14 * radius ** 2
print('Area of circle: ', area)

print('Circle radius is {} and area is {}'.format(radius,area))
```

Today we will cover

- *** Introduction to python - done***
- *** Basic Data types -done***
- *** Strings -done***
- set, Tuples, Dictionaries
- Operators
- Conditional statements

Lists Tuples Sets and Dictionaries

We will be discussing all the basics required in Python for this Project.

Please note that this is a Completely Research Based Project, we will cover only all the required things which you need to know in order to complete the project and direct you towards the resources in case if you are interested in detailed study of topics

We will follow this notebook for this session

This notebooks will go through the basic topics in order as mentioned:

- Built-in Python collections:
  - Lists: List is created using square brackets or list() method. Ordered, Mutable, allows duplicates.
  - Tuples: Tuple created using small brackets or tuple() method. Ordered, immutable, allows duplicates.
  - sets: Set created using set() method. Unordered, mutable, doesn't allow duplicates.
  - Dictionaries: Dictionary created using flower brackets or dict() method. Ordered, Mutable, doesn't allow duplicates.

- Numeric Literals or String Literals that we have seen till now accept only one data type.
- There are other built in data types in python that store collection of data.

List

If you are familiar with another programming language, start to draw parallels between lists in Python and arrays in other language. There are two reasons which tells why the lists in Python are more flexible than arrays in other programming language:

- They have no fixed size (which means we need not to specify how big the list will be)
- They have no fixed type constraint

Earlier, while discussing introduction to strings we have introduced the concept of a *sequence* in Python.

In Python, Lists can be considered as the most general version of a "sequence". Unlike strings, they are mutable which means the elements inside a list can be changed!

Lists are constructed with brackets [] and commas separating every element in the list.

Let's go ahead and see how we can construct lists!

1 - Dimensional data structure / container

There is no limitation on data type - a list can store integer, string, boolean and so on

Every data type is preserved - there is no forced type casting that happens

Create list and view them

Create using [] and list(). View using name and print()

```
# creation of empty list
mylist = []
print(mylist)

mylist = list()
print(mylist)
```

```
[]
[]
```

```
# Creation of list with values
my_list = [1,2,3]
print(my_list)
```

```
my_list1 = list((1,2,3))
my_list1
```

```
[1, 2, 3]
[1, 2, 3]
```

```
# We just created a list of integers, but lists can actually hold different object types.
my_list2 = ['A string',23,100.232,True]
print(my_list2)

# Nested lists are allowed too. In fact List can have tuples, dictionaries and sets within
nested_list = [[1,2,3],(1,2,3),{1:'first',2:'sec'},{1,2,3}]
print(nested_list)
```

```
['A string', 23, 100.232, True]
[[1, 2, 3], (1, 2, 3), {1: 'first', 2: 'sec'}, {1, 2, 3}]
```

```
# Built-in Len() funtion of python can be applied on list to ge number of elements in it
len(nested_list)
```

```
4
```

Access list using Indexing and Slicing

Indexing and slicing of lists works just like in Strings. Let's make a new list to remind ourselves of how this works:

```
# Accessing the list, Indexing starts from 0 in python
mylist = [10,10.3,"Abc", "Spartificial",True]

# Access single element using index
mylist[0]
```

```
10
```

```
# Retrive a elements using slicing [index_start, Index_end, step].
# Ending element will be from index_end specified - 1.

#get all elements from position 1 till end
print(mylist[1:])

# get elements from position 1 till position 3
print(mylist[1:3])

# Get two elements from start
print(mylist[:2])

# get elements in even positions
print(mylist[::2])
```

```
# get elements in even positions in reverse order
print(mylist[::-2])
```

```
[10.3, 'Abc', 'Spartificial', True]
[10.3, 'Abc']
[10, 10.3]
[10, 'Abc', True]
[True, 'Abc', 10]
```

```
mylist
```

```
[10, 10.3, 'Abc', 'Spartificial', True]
```

Add elements to list

- Lists are ordered (the order in which the elements are added is retained) and mutable (being able to add elements to list after creation)

```
# Repeating the list elements n times
mylist * 3
```

```
[10,
 10.3,
 'Abc',
 'Spartificial',
 True,
 10,
 10.3,
 'Abc',
 'Spartificial',
 True,
 10,
 10.3,
 'Abc',
 'Spartificial',
 True]
```

```
# Concatenation in list

# We can also use "+" to concatenate lists, just like we did for Strings.

mylist + ['new item',5]
```

```
[10, 10.3, 'Abc', 'Spartificial', True, 'new item', 5]
```

```
mylist
```

```
[10, 10.3, 'Abc', 'Spartificial', True]
```

Note:

Any operations on the list will not affect the original list. To make the changes in the original list we have to reassign the list

```
# concatenate two lists similar to string concatenation and reassign
mylist = mylist + ['new item',5]
mylist
```

```
[10, 10.3, 'Abc', 'Spartificial', True, 'new item', 5]
```

add elements using List methods:

When list methods are used to make changes to list they impact the original list

```
# Use extend method to concatenate the lists
list1 =['apple', 'banana', 'orange']
list2=['watermelon', 'grapes']
list1.extend(list2)
list1
```

```
['apple', 'banana', 'orange', 'watermelon', 'grapes']
```

```
# Use the .append() method to permanently add an item to the end of a list:
mylist = ['apple','banana','orange']
mylist.append('grapes')
mylist
```

```
['apple', 'banana', 'orange', 'grapes']
```

```
# Insert element in between
print(mylist[2])
mylist.insert(2,'watermelon')
print(mylist)
```

```
orange
['apple', 'banana', 'watermelon', 'orange', 'grapes']
```

Modify or remove elements in list

```
# Change the element value by using index of element
print(mylist[2])
mylist[2] = 'strawberry'
print(mylist)
```

```
watermelon
['apple', 'banana', 'strawberry', 'orange', 'grapes']
```

```
# use .pop() method to remove last element from the list
mylist.pop()
mylist
```

```
['apple', 'banana', 'strawberry', 'orange']
```

```
# use Pop to remove element from a particular index and returns element
# remove element @ 0
mylist.pop(0)
```

```
'apple'
```

```
print(mylist)
```

```
['banana', 'strawberry', 'orange']
```

```
# use del to remove elements from list using index but element is not returned
del mylist[2]
```

```
print(mylist)
```

```
['banana', 'strawberry']
```

```
# use remove() to remove elements from list using index but element is not returned
mylist.remove('banana')
print(mylist)
```

```
['strawberry']
```

```
# empty the list removing all elements
mylist.clear()
print(mylist)
```

```
[]
```

Miscellaneous methods

```
mylist = ['apple','banana','chikoo','apple','banana','banana']

# get count of an element
print('count of apple: ',mylist.count('apple'))

# Copy elements from list
copy_list = mylist.copy()
print(copy_list)

# Provides index of first occurence of element
print('index of apple in list:', mylist.index('apple'))

# Reverse the list
mylist.reverse()
print('Reversed list: ', mylist)
```

```
#sort the list
mylist.sort()
print('sorted list: ', mylist)
```

```
count of apple:  2
['apple', 'banana', 'chikoo', 'apple', 'banana', 'banana']
index of apple in list: 0
Reversed list:  ['banana', 'banana', 'apple', 'chikoo', 'banana', 'apple']
sorted list:  ['apple', 'apple', 'banana', 'banana', 'banana', 'chikoo']
```

```
# List all available methods for list objects
help(list)
```

```
Help on class list in module builtins:

class list(object)
 |  list(iterable=(), /)
 |
 |  Built-in mutable sequence.
 |
 |  If no argument is given, the constructor creates a new empty list.
 |  The argument must be an iterable if specified.
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __delitem__(self, key, /)
 |      Delete self[key].
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getitem__(...)
 |      x.__getitem__(y) <==> x[y]
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __iadd__(self, value, /)
 |      Implement self+=value.
 |
 |  __imul__(self, value, /)
 |      Implement self*=value.
 |
 |  __init__(self, /, *args, **kwargs)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
```

```
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __len__(self, /)
 |      Return len(self).
 |
 |  __lt__(self, value, /)
 |      Return self<value.
 |
 |    mul  (self, value, /)
```

```
dir(list)
```

```
['__add__',
 '__class__',
 '__contains__',
 '__delattr__',
 '__delitem__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__gt__',
 '__hash__',
 '__iadd__',
 '__imul__',
 '__init__',
 '__init_subclass__',
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
 '__mul__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__reversed__',
 '__rmul__',
 '__setattr__',
 '__setitem__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 'append',
 'clear',
 'copy',
 'count',
 'extend',
 'index',
 'insert',
 'pop',
 'remove',
```

```
        'reverse',
        'sort']
```

Comprehension:

Comprehension in python is a way to construct new sequences using short and concise instructions. Lets see how to use comprehension on lists to construct new lists.

```
# Without comprehension create a list with squares of numbers from 1 to 10

list1=[]
for i in range(1,11):
  list1.append(i**2)
else:
  print('list with squares of numbers from 1-10: ', list1)
```

```
    list with squares of numbers from 1-10:  [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

```
# With comprehension create a list with squares of numbers from 1 to 10

list1=[i**2 for i in range(1,11)]

print('list with squares of numbers from 1-10: ', list1)
```

```
    list with squares of numbers from 1-10:  [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Tuples

In Python, tuples are similar to lists but they are immutable i.e. they cannot be changed. You would use the tuples to present data that shouldn't be changed, 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
3.) When to use tuples
```

You'll have an intuition of how to use tuples based on what you've learned about lists. But, Tuples work very similar to lists but the major difference is tuples are immutable.

Constructing Tuples

The construction of tuples use () with elements separated by commas where in the arguments will be passed within brackets. For example:

Create tuple and view them

Create using () and tuple(). View using name and print()

```
# creation of empty tuple
mytuple = ()
print(mytuple)

mytuple = tuple()
print(mytuple)
```

```
()
()
```

```
# Creation of tuple with values
my_tuple = (1,2,3)
print(my_tuple)

my_tuple1 = tuple((1,2,3))
my_tuple1
```

```
(1, 2, 3)
(1, 2, 3)
```

```
# We just created a tuple of integers, but tuples can actually hold different object types
my_tuple = ('A string',23,100.232,True)
print(my_tuple)

# Nested tuples are allowed too. In fact tuple can have lists, dictionaries and sets withi
nested_tuple = ([1,2,3],(1,2,3),{1:'first',2:'sec'},{1,2,3})
print(nested_tuple)
```

```
('A string', 23, 100.232, True)
([1, 2, 3], (1, 2, 3), {1: 'first', 2: 'sec'}, {1, 2, 3})
```

```
# Built-in Len() funtion of python can be applied on tuple to ge number of elements in it
len(nested_tuple)
```

```
4
```

Access tuple using Indexing and Slicing

Indexing and slicing of tuples works just like in tuples. Let's make a new tuple to remind ourselves of how this works:

```
# Accessing the tuple, Indexing starts from 0 in python
mytuple = (10,10.3,"Abc", "Spartificial",True)

# Access single element using index subscripting
mytuple[0]
```

```
10
```

```
# Retrive a elements using slicing [index_start, Index_end, step].
# Ending element will be from index_end specified - 1.

#get all elements from position 1 till end
print(mytuple[1:])

# get elements from position 1 till position 3
print(mytuple[1:3])

# Get two elements from start
print(mytuple[:2])

# get elements in even positions
print(mytuple[::2])

# get elements in even positions in reverse order
print(mytuple[::-2])
```

```
(10.3, 'Abc', 'Spartificial', True)
(10.3, 'Abc')
(10, 10.3)
(10, 'Abc', True)
(True, 'Abc', 10)
```

```
mytuple
```

```
(10, 10.3, 'Abc', 'Spartificial', True)
```

Add elements to tuple

- tuples are ordered (the order in which the elements are added is retained) and immutable (not able to add elements to tuple after creation)

```
# Repeating the tuple elements n times
mytuple * 3
```

```
(10,
 10.3,
 'Abc',
 'Spartificial',
 True,
 10,
 10.3,
```

```
        'Abc',
        'Spartificial',
        True,
        10,
        10.3,
        'Abc',
        'Spartificial',
        True)
```

```
# Concatenation in tuple

# We can also use "+" to concatenate tuples, just like we did for Strings.

mytuple + ('new item',5)
```

```
(10, 10.3, 'Abc', 'Spartificial', True, 'new item', 5)
```

```
mytuple
```

```
(10, 10.3, 'Abc', 'Spartificial', True)
```

```
# To show that tuples are indeed immutable
#mytuple[2] = 'ABC'
mytuple.insert('abc')
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-39-9fd9626a7282> in <module>()
      1 # To show that tuples are indeed immutable
      2 #mytuple[2] = 'ABC'
----> 3 mytuple.insert('abc')

AttributeError: 'tuple' object has no attribute 'insert'
```

SEARCH STACK OVERFLOW

Note:

No operations like add or insert or remove can be made to tuples as they are immutable. Concatenation of tuples is possible though.

```
# concatenate two tuples similar to string concatenation and reassign
mytuple = mytuple + ('new item',5)
mytuple
```

```
(10, 10.3, 'Abc', 'Spartificial', True, 'new item', 5)
```

Miscellaneous methods

Only count() and index() methods are available

```
mytuple = ('apple','banana','chikoo','apple','banana','banana')

# get count of an element
print('count of apple: ',mytuple.count('apple'))

# Provides index of first occurence of element
print('index of apple in tuple:', mytuple.index('apple'))
```

```
count of apple:  2
index of apple in tuple: 0
```

```
# List all available methods for tuple objects
help(tuple)
```

```
Help on class tuple in module builtins:

class tuple(object)
 |  tuple(iterable=(), /)
 |
 |  Built-in immutable sequence.
 |
 |  If no argument is given, the constructor returns an empty tuple.
 |  If iterable is specified the tuple is initialized from iterable's items.
 |
 |  If the argument is a tuple, the return value is the same object.
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getitem__(self, key, /)
 |      Return self[key].
 |
 |  __getnewargs__(self, /)
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __hash__(self, /)
 |      Return hash(self).
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
```

```
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __len__(self, /)
 |      Return len(self).
 |
 |  __lt__(self, value, /)
 |      Return self<value.
 |
 |  __mul__(self, value, /)
 |      Return self*value.
 |
 |  __ne__(self, value, /)
 |      Return self!=value.
 |
```

```
dir(tuple)
```

```
['__add__',
 '__class__',
 '__contains__',
 '__delattr__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__getnewargs__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
 '__mul__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__rmul__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 'count',
 'index']
```

Comprehension:

Comprehension in python is a way to construct new sequences using short and concise instructions. Lets see how to use comprehension on tuples to construct new tuples.

```
# With comprehension create a list with squares of numbers from 1 to 10

tuple1=tuple(i**2 for i in range(1,11))

print('tuple with squares of numbers from 1-10: ', tuple1)
```

```
tuple with squares of numbers from 1-10:  (1, 4, 9, 16, 25, 36, 49, 64, 81, 100)
```

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. While you are passing around an object and if you need to make sure that it does not get changed then tuple become your solution. It provides a convenient source of data integrity.

You should now be able to create and use tuples in your programming as well as have a complete understanding of their immutability.

```
# Usage of some built-in python functions on tuple, we have already seen len()
t=("c","c++","c#","java", "python")

# Get smallest element from tuple
print(max(t))

# Get largest element from tuple
print(min(t))

# Use of membership operator to check if the element is present in tuple
print("python" in t)

# tuple doesn't have sort method as tuple is immutable but python sorted() method can be u
print(sorted(t))
```

```
python
c
True
['c', 'c#', 'c++', 'java', 'python']
```

```
# Immutable is applied only on tuple if the object is tuple of lists, list is mutable.
matrix=(['00','01','02'],[10,11,12],[20,21,22])
matrix[0][0]=100
print(matrix)

matrix[0][1]=[100,111,222]
print(matrix)
```

```
([100, '01', '02'], [10, 11, 12], [20, 21, 22])
([100, [100, 111, 222], '02'], [10, 11, 12], [20, 21, 22])
```

Sets

Sets are an unordered collection of unique elements which can be constructed using the set() function. Sets are used to store multiple items in a single variable. A set is a collection which is unordered, unchangeable (New elements can be added and exisiting elements can be removed), and unindexed.

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

Create set and view them

Create using set(). View using name and print().

```
# creation of empty set
myset = set()
print(myset)

# Note that we are using only set methods but not {} as they are resevered for dictionarie
```

```
set()
```

```
# Creation of set with values
my_set = {1,2,3}
print(my_set)

my_set1 = set((1,2,3))
my_set1
```

```
{1, 2, 3}
{1, 2, 3}
```

```
# We just created a set of integers, but sets can actually hold different object types.
my_set2 = {'A string',23,100.232,True}
print(my_set2)

# Nested sets are allowed too. In fact set can have tuples, dictionaries and sets within t
nested_set = {(1,2,3),(5,4,6)} #unordered example, also immutable objects are hashable and
print(nested_set)
```

```
{'A string', True, 100.232, 23}
{(5, 4, 6), (1, 2, 3)}
```

```
# Built-in Len() funtion of python can be applied on set to get number of elements in it
len(nested_set)
```

```
2
```

Access set using Indexing and Slicing

Set elements can't be accessed using index and subscripting, hence slicing operations are not allowed too

```
# Accessing the set, Indexing starts from 0 in python
myset = {10,10.3,"Abc", "Spartificial",True}

# Set objects an't accessed using subscripts
myset[0]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-31-aa97aa6f14a3> in <module>()
      3
      4 # Set objects an't accessed using subscripts
----> 5 myset[0]

TypeError: 'set' object is not subscriptable
```

SEARCH STACK OVERFLOW

```
# We can traverse through list using loops
for i in myset:
  print(i)
```

```
True
Abc
10
10.3
Spartificial
```

Add elements

Elements can be added to set only to the end. Since sets are unindexed we can't change the elements that are already present

```
myset={6,2,3,4}

#Appends elements to set
myset.add(5)
myset
```

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

Note that the curly brackets do not indicate a dictionary! Using only keys, you can draw analogies as a set being a dictionary.

We know that a set has an only unique entry. Now, let us see what happens when we try to add something more that is already present in a set?

```
# Add a different element
myset.add(1)
myset
```

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

Note:

Sets cannot be concatenated like lists using '+'

```
# Use update() to add elements to set
myset = {1,2,3,4}
mytuple = (5,6,7,8)

# Note that the update sequence can be list, tuple or set
myset.update(mytuple)
myset
```

```
{1, 2, 3, 4, 5, 6, 7, 8}
```

Modify or remove elements in list

```
# use .pop() method to remove last element from the list
myset.pop()
myset
```

```
{2, 3, 4, 5, 6, 7, 8}
```

```
#Note that we can't pop elements from set by giving index as sets are unindexed
```

```
# use remove() to remove elements from set
myset.remove(3)
print(myset)
```

```
{2, 4, 5, 6, 7, 8}
```

```
# use discard() to remove elements from set
myset.discard(5)
print(myset)
```

```
{2, 4, 6, 7, 8}
```

```
# only difference between discard and remove is that no error is raised for discard() when
print(myset.discard(1))

print(myset.remove(1))
```

```
None
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-59-60a5d1ac928a> in <module>()
      2 print(myset.discard(1))
      3
----> 4 print(myset.remove(1))

KeyError: 1
```

SEARCH STACK OVERFLOW

```
# empty the list removing all elements
myset.clear()
print(myset)
```

```
set()
```

Miscellaneous methods

```
myset = {'apple','banana','chikoo'}

# Copy elements from list
copy_set = myset.copy()
print(copy_set)
```

```
{'apple', 'chikoo', 'banana'}
```

```
set1 = {'a','b','c','d','g'}
set2 = {'a','b','e','f','g'}

# All the below methods make no change to original sets
# union() provides super set of 2 sets
print(set1.union(set2))

# difference() provides elements only in set1
print(set1.difference(set2))

# intersection() provides common elements from set1 and set
print(set1.intersection(set2))

# symmetric_difference() provides unique elements from set1 and set
print(set1.symmetric_difference(set2))
```

```
{'d', 'b', 'f', 'c', 'g', 'e', 'a'}
{'c', 'd'}
{'g', 'b', 'a'}
{'e', 'c', 'd', 'f'}
```

Note:

The set operations can alternatively be performed using

- minus (-) for difference. set1 - set2
- and (&) for interestion. set1 & set2
- or (^) for union. set1 ^ set2

```
# All the below methods change the original sets

# difference_update() provides elements only in set1
set1= {'a','b','c','d','g'}
set1.difference_update(set2)
print(set1)

# intersection_update() provides common elements from set1 and set
set1={'a','b','c','d','g'}
set1.intersection_update(set2)
print(set1)

# symmetric_difference_update() provides unique elements from set1 and set
set1={'a','b','c','d','g'}
set1.symmetric_difference_update(set2)
print(set1)
```

```
{'d', 'c'}
{'g', 'b', 'a'}
{'d', 'f', 'c', 'e'}
```

```
# Methods that return boolean values
set1={'a','c','d','e','f','g'}
set2={'f','g'}
#set2={'f','g','h'}
set3={'h'}

# Set1 is said to be superset of set2 if all the elements in set2 are available in set1. s
print('set1 is superset of set2') if set1.issuperset(set2) else print('set1 is not superse

print('set2 is subset of set1') if set2.issubset(set1) else print('set2 is not subset of s

# two sets are said to be disjoint if neither of them have common elements
print(set3.isdisjoint(set2))
```

```
set1 is superset of set2
set2 is subset of set1
True
```

```
# List all available methods for set objects
help(set)
```

```
Help on class set in module builtins:

class set(object)
 |  set() -> new empty set object
```

```
|  set(iterable) -> new set object
|
|  Build an unordered collection of unique elements.
|
|  Methods defined here:
|
|  __and__(self, value, /)
|      Return self&value.
|
|  __contains__(...)
|      x.__contains__(y) <==> y in x.
|
|  __eq__(self, value, /)
|      Return self==value.
|
|  __ge__(self, value, /)
|      Return self>=value.
|
|  __getattribute__(self, name, /)
|      Return getattr(self, name).
|
|  __gt__(self, value, /)
|      Return self>value.
|
|  __iand__(self, value, /)
|      Return self&=value.
|
|  __init__(self, /, *args, **kwargs)
|      Initialize self.  See help(type(self)) for accurate signature.
|
|  __ior__(self, value, /)
|      Return self|=value.
|
|  __isub__(self, value, /)
|      Return self-=value.
|
|  __iter__(self, /)
|      Implement iter(self).
|
|  __ixor__(self, value, /)
|      Return self^=value.
|
|  __le__(self, value, /)
|      Return self<=value.
|
|  __len__(self, /)
|      Return len(self).
|
|  __lt__(self, value, /)
|      Return self<value.
|
|  __ne__(self, value, /)
|      Return self!=value.
|
```

```
dir(set)
```

```
['__and__',
 '__class__',
```

```
'__contains__',
'__delattr__',
'__dir__',
'__doc__',
'__eq__',
'__format__',
'__ge__',
'__getattribute__',
'__gt__',
'__hash__',
'__iand__',
'__init__',
'__init_subclass__',
'__ior__',
'__isub__',
'__iter__',
'__ixor__',
'__le__',
'__len__',
'__lt__',
'__ne__',
'__new__',
'__or__',
'__rand__',
'__reduce__',
'__reduce_ex__',
'__repr__',
'__ror__',
'__rsub__',
'__rxor__',
'__setattr__',
'__sizeof__',
'__str__',
'__sub__',
'__subclasshook__',
'__xor__',
'add',
'clear',
'copy',
'difference',
'difference_update',
'discard',
'intersection',
'intersection_update',
'isdisjoint',
'issubset',
'issuperset',
'pop',
'remove',
'symmetric_difference',
'symmetric_difference_update',
'union',
'update']
```

Comprehension:

Comprehension in python is a way to construct new sequences using short and concise instructions. Lets see how to use comprehension on sets to construct new sets.

```
# With comprehension create a set with squares of numbers from 1 to 10

set1={i**2 for i in range(1,11)}

print('set with squares of numbers from 1-10: ', set1)
```

```
set with squares of numbers from 1-10:  {64, 1, 4, 36, 100, 9, 16, 49, 81, 25}
```

Dictionaries

These dictionaries are nothing but hash tables in other programming languages. The data is stored in dictionary in key:value pairs. They are ordered (Retain the order in which the elements are inserted), mutable (Allows edit operations) and allows duplicates. The dictionaries are indexed by key like lists they are not indexed by position.

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 objects by their relative position. This is an important distinction since mappings won't retain the order since they have objects defined by a key.

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

Constructing a Dictionary

Let's see how we can construct dictionaries to get a better understanding of how they work!

Create dictionary and view them

Create using {} and dict(). View using name and print()

```
# creation of empty list
mydict = {}
print(mydict)

mydict = dict()
```

```
print(mydict)
```

```
{}
{}
```

```
# The dictionary stores key value pairs, even the data should be provided in same manner
my_dict = {1:'yash',2:'aman',3:'animesh',4:'sai'}
print(my_dict)

key=[1,2,3,4]
value=['yash','aman','animesh','sai']
my_dict1 = dict(zip(key,value))
my_dict1
```

```
{1: 'yash', 2: 'aman', 3: 'animesh', 4: 'sai'}
{1: 'yash', 2: 'aman', 3: 'animesh', 4: 'sai'}
```

```
# We just created a list of integers, but lists can actually hold different object types.
my_dict2 = {1:'A string','a':23,3.85:100.232,2:True}
print(my_dict2)

# Nested dicts are allowed too. In fact List can have tuples, dictionaries and sets within
nested_dict = {1: {'name': 'John', 'age': '27', 'sex': 'Male'},
          2: {'name': 'Marie', 'age': '22', 'sex': 'Female'}}
print(nested_dict)
```

```
{1: 'A string', 'a': 23, 3.85: 100.232, 2: True}
{1: {'name': 'John', 'age': '27', 'sex': 'Male'}, 2: {'name': 'Marie', 'age': '22',
```

```
# Built-in Len() funtion of python can be applied on dict to get number of elements in it
len(nested_dict)
```

```
2
```

Access dictionary using keys

Since dictionaries are key value pairs we can get values from dictionary using keys

```
# Accessing the dict, Indexing starts from 0 in python
mydict = {1: {'name': 'John', 'age': '27', 'sex': 'Male'}, 2: {'name': 'Marie', 'age': '22

# Access single dictionaries using key
print(mydict[1])

print(mydict[1]['name'])
```

```
{'name': 'John', 'age': '27', 'sex': 'Male'}
John
```

Add elements to list

- Lists are ordered (the order in which the elements are added is retained) and mutable (being able to add elements to list after creation)

```
mydict = {'sai':10,'yash':20,'aman':30}

# Use update method to add key:value pairs to dictionary. If the key is not available the
mydict.update({'sai':10})
print(mydict)

mydict.update({'somu':40,'sreshta':50,'shrenik':60})
print(mydict)
```

```
{'sai': 10, 'yash': 20, 'aman': 30}
{'sai': 10, 'yash': 20, 'aman': 30, 'somu': 40, 'sreshta': 50, 'shrenik': 60}
```

Modify or remove elements in list

```
# Change the element value by using key of element
mydict.update({'sai':80})
print(mydict)
```

```
{'sai': 80, 'yash': 20, 'aman': 30, 'somu': 40, 'sreshta': 50, 'shrenik': 60}
```

```
# Remove key:value pair from dictionary using key
mydict.pop('sai')
```

```
80
```

```
print(mydict)
```

```
{'yash': 20, 'aman': 30, 'somu': 40, 'sreshta': 50, 'shrenik': 60}
```

```
# Remove last item from the dictionary
mydict.popitem()
```

```
('shrenik', 60)
```

```
print(mydict)
```

```
{'yash': 20, 'aman': 30, 'somu': 40, 'sreshta': 50}
```

```
# Empties the dictionary
mydict.clear()
print(mydict)
```

```
{}
```

Miscellaneous methods

```
mydict = {'sai':10,'yash':20,'aman':30}

# Copy elements from dictionary
copy_dict = mydict.copy()
print(copy_dict)

# Get keys from dictionary
print(mydict.keys())

# Get values from dictionary
print(mydict.values())

# Get items from dictionary
print(mydict.items())
```

```
{'sai': 10, 'yash': 20, 'aman': 30}
dict_keys(['sai', 'yash', 'aman'])
dict_values([10, 20, 30])
dict_items([('sai', 10), ('yash', 20), ('aman', 30)])
```

```
# List all available methods for list objects
help(list)
```

```
Help on class list in module builtins:

class list(object)
 |  list(iterable=(), /)
 |
 |  Built-in mutable sequence.
 |
 |  If no argument is given, the constructor creates a new empty list.
 |  The argument must be an iterable if specified.
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __delitem__(self, key, /)
 |      Delete self[key].
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
```

```
|  __getitem__(...)
|      x.__getitem__(y) <==> x[y]
|
|  __gt__(self, value, /)
|      Return self>value.
|
|  __iadd__(self, value, /)
|      Implement self+=value.
|
|  __imul__(self, value, /)
|      Implement self*=value.
|
|  __init__(self, /, *args, **kwargs)
|      Initialize self.  See help(type(self)) for accurate signature.
|
|  __iter__(self, /)
|      Implement iter(self).
|
|  __le__(self, value, /)
|      Return self<=value.
|
|  __len__(self, /)
|      Return len(self).
|
|  __lt__(self, value, /)
|      Return self<value.
|
|  __mul__(self, value, /)
```

```
dir(dict)
```

```
['__class__',
 '__contains__',
 '__delattr__',
 '__delitem__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__setitem__',
 '__sizeof__',
 '__str__',
```

```
 '__subclasshook__',
 'clear',
 'copy',
 'fromkeys',
 'get',
 'items',
 'keys',
 'pop',
 'popitem',
 'setdefault',
 'update',
 'values']
```

Comprehension:

Comprehension in python is a way to construct new sequences using short and concise instructions. Lets see how to use comprehension on lists to construct new lists.

```
# With comprehension create a list with squares of numbers from 1 to 10

mydict={i:i**2 for i in range(1,11)}

print('dict with squares of numbers from 1-10: ', mydict)
```

```
dict with squares of numbers from 1-10:  {1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 4
```

Double-click (or enter) to edit

Today we will cover

- *** Introduction to python - done***
- *** Basic Data types -done***
- *** Strings -done***
- *** List comprehension -done***
- *** set, Tuples, Dictionaries - done***
- Operators

Comparison Operators

```
a = 2
b = 10
```

```
a  > b
```

```
False
```

```
a < b
```

```
True
```

```
a == b
```

```
False
```

```
a != b
```

```
True
```

```
a >= b
```

```
False
```

```
a <= b
```

```
True
```

```
'hi' == 'bye'
```

```
False
```

Logic Operators

```
(1 > 2) and (2 < 3)
```

```
False
```

```
(1 == 2) or (2 == 3) or (4 == 4)
```

```
True
```

```
(1 > 2) or (2 < 3)
```

```
True
```

```
# Membership operators -- in / not in
"Animesh" in [10,20,30,40,50,"Animesh"]
```

```
True
```