# Lecture 1,2,3 (Video 1):

## Naming and Using Variables:

* Variables names can contain only letters, numbers and underscores
* Spaces are not allowed in variable names, but underscores can be used to separate words in variable names
* Avoid using Python keywords and function names as variable names
* Variable names should be short but discriptive.

A traceback is the record of where the interprator ran into trouble while executing the code.

## Strings

* You can use double as well as single quotes like,
  + ‘This is a “new line”’
  + “This is a ‘new line’”
* Here are some methods provided by python to remove whitespace from variable or message
  + lstrip() - removes whitespace from left
  + rstrip() - removes whitespace from right
  + strip() - removes whitespace from both sides
* To assign a variable’s value into new variable or print() use f at the starting and {variable’s name} in parenthesis
* When you’re writing long numbers, you can group digits using underscores to make a large numbers more readable:
* >>> age = 14\_000\_000\_000  
   >>> print(age)  
   14000000000
* You can assign values to more than one variable using just a single line: python >>> x, y, z = 0,9,3
* Python doesn’t have built-in constant types, but Python programmers use all capital letters to indicate a variable should be treated as a constant and never be changed.

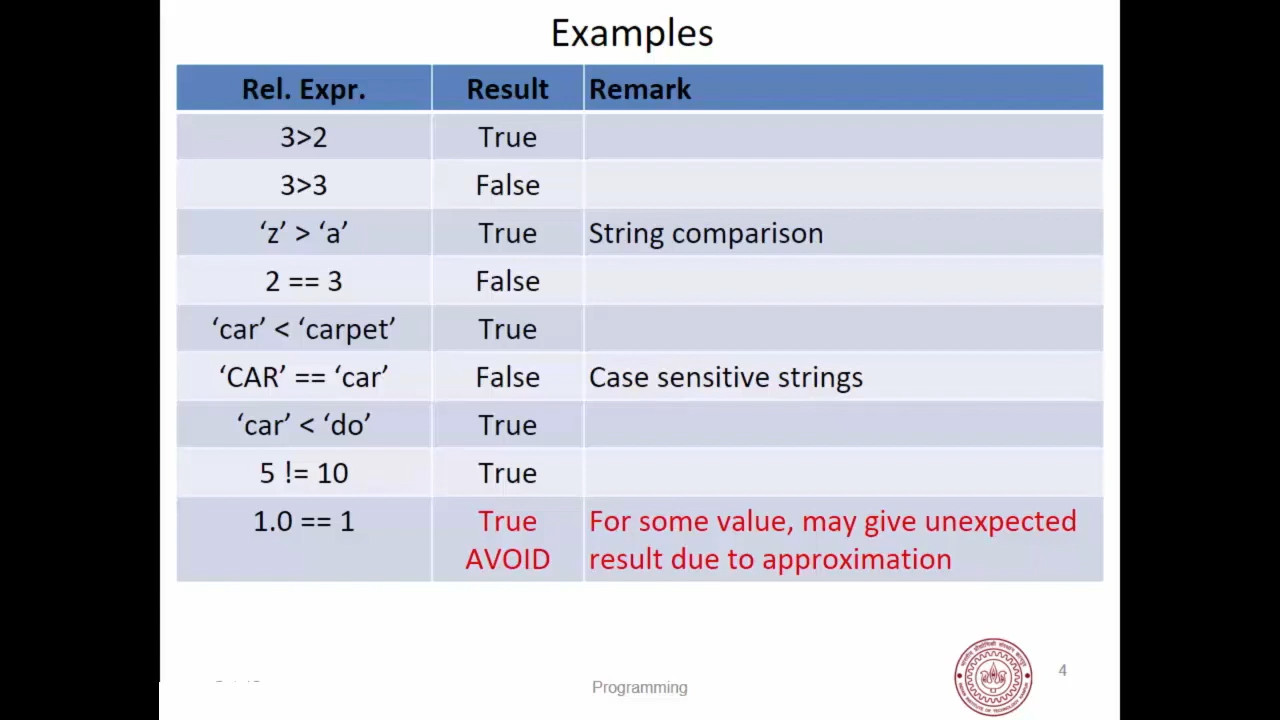
## Elements of Python:

* A python program is a sequence of definations and commands(statement)
* Commands manipulate objects
* Each object is associated with a Type
* Type:
  + A set of values
  + A set of operations on these values
* Expressions: An operation (combination of objects and operators)

### Types in Python:

* int:
  + Bounded integers, e.g., 732 or -5
* float:
  + Real numbers, e.g., 3.14
* long:
  + long integers with unlimited precision
* str:
  + Strings, e.g., ‘hello’ or ‘c’
* two types in python:
  + Scalar:
    - Individual objects that don’t have internal structure
    - int(signed integers), float(floating point), bool(Boolean), NoneType
      * NoneType is special type with single value which is None
  + Non-Scalar:
    - Objects have internal structure
    - str(strings)
* You can use type function to find the type of an expression
* Note that, float to int conversion is truncation not rounding off

## Recap

* Python program is a sequence of commands
* Commands manipulate objects
* Objects have types
* Commmands contain operations:
  + Arithmetic, logical, relational,…
* Multiple operations are possible in a single command
*  # Lecture 4:
* Most lists you create will be dynamic
  + Means build a list and then add or remove elements from it as your program runs
* The simplest way to add a new item to a list is to append the item to the list.

## Organizing the list:

Often time we don’t have control of how the elements are entered in the list. So, to organize them we have few methods:

### Sorting a List Permanantly with sort() Method:

* Python’s sort() method makes it relatively easy to sort a list.
* You can also sort any list in reverse alphabetical order by passing the argument reverse=True to sort() method.
* syntax of sort is: list\_name.sort(key=NONE(use some), reverse=Boolean\_value) > It is not possible to revert back to orignal order now.

### Sorting a List Temporarily with sorted() Function:

* Maintain the original order of a list but present it in a sorted order, use sorted() function.
* syntax of sorted is: sorted(list\_name, key=NONE(use some), reverse=Boolean\_value)

### Printing a List in Reverse Order:

* To reverse the original order of a list, you can use the reverse() method.
* The reverse() method changes the order of a list permanantly, but you can revert to the original order anytime by applying reverse() to the same list second time.

## Finding the Length of a List:

* We can quickly find the length of a list by using the len() function
* We’ll find len() useful when you need to identify the number of aliens that still to be shot down in a game.

# Lecture 6:

## Tuples:

A tuple consists of a number of values separated by commas

>>> t = 'abhay', 'shanker', 'pathak', 101  
>>> t[2]  
'pathak'  
>>> type(t)  
<class, 'tuple'>  
>>> empty = ()  
>>> type(empty)  
<class, 'tuple'>  
>>> single = 1,  
>>> type(single)  
<class, 'tuple'>  
>>> single  
1,  
>>> single = 1  
>>> type(single)  
<class, 'int'>

As, you can see above to get a single tuple comma after that single element is necessary

### Nesting tuples:

Tuples can be nested in following way:

>>> course = 'python', 'lecturer', 101  
>>> student = 'you', 20, course  
# I nested `course` tuple inside `student` tuple  
>>> course  
('python', 'lecturer', 101)  
>>> student  
('you', 20, ('python', 'lecturer', 101))

Important thing is that course tuple is copied to student tuple, not like embedde. So, If we do something like this

>>> course = 'python', 'lecturer', 102  
>>> course  
('python', 'lecturer', 102)  
>>> student  
('you', 20, ('python', 'lecturer', 101))

As you can see, we previously made course a tuple element of student, then after changing course it’s still the same. So, it copies. Changing course doesn’t affects student.

### length of tuple:

Using tuples from previous headings

>>> len(singleton)  
1  
>>> len(empty)  
0  
>>> len(student)  
3  
# it'll not show 5, cause it's treating course as a element  
>>> len(student + course)  
6

### more operation on tuples:

Tuples can be **concatenated, repeated, indexed and sliced**. Here’s some examples:

* **Concatenation**:

>>> student + course  
('abhay shanker', 45, ('c', 'shivam', 102), 'c', 'shivam', 103)  
>>> (course + student)[3]  
'abhay shanker'  
>>> (student + course)[2:]  
(('c', 'shivam', 102), 'c', 'shivam', 103)  
>>> (course + student)[1:6]  
>>> (course + student)[1:6]  
('shivam', 103, 'abhay shanker', 45, ('c', 'shivam', 102))  
>>> (course + student)[1:5]  
('shivam', 103, 'abhay shanker', 45)

* **repeatition**:

>>> 2\*student  
('abhay shanker', 45, ('c', 'shivam', 102), 'abhay shanker', 45, ('c',  
'shivam', 102))

## Unpacking Sequences:

* Strings and tuples are examples of sequences:
  + indexing, slicing, concatenation, repeatition operations applicable on seqsequences.
* Sequence unpacking operation can be applied to sequences to get the components:
  + *Mulitple assignment statement*
  + LHS and RHS must have equal length Let’s take examples:

>>> student # from previous tuple  
('abhay shanker', 45, ('c', 'shivam', 102))  
>>> name, roll, course = student  
>>> name  
'abhay shanker'  
>>> roll  
45  
>>> course  
('c', 'shivam', 102)  
# let's take string as example as it is also sequence  
>>> a,b,c='dhanu'  
Traceback (most recent call last):  
 File "<input>", line 1, in <module>  
 a,b,c='dhanu'  
ValueError: too many values to unpack (expected 3)  
>>> a,b,c = 'luv'  
>>> a  
'l'  
>>> b  
'u'  
>>> c  
'v'  
>>> print(a,b,c)  
l u v

As you have noticed from above that, LHS should equal to RHS

## Lists

* Ordered sequence of values
* Written as sequence of comma-separated values between square brackets
* Values can be of different types:
  + usually the items all have the same types

>>> lst = [1, 2, 3]  
>>> type(lst)  
<class 'list'>  
>>> lst1 = [2, 3, 'name']  
>>> lst1  
[2, 3, 'name']

* List is also a sequence type:
  + All the operations of sequence are also applicable here

>>> len(lst)  
3  
>>> lst1[1:] # slicing  
[3, 'name']  
>>> [0] + lst1 + [10] # concatenation  
[0, 2, 3, 'name', 10]  
>>> 3 \* lst # repeatition  
[1, 2, 3, 1, 2, 3, 1, 2, 3]  
>>> x, y, z = lst1 # unpacking  
>>> x  
2  
>>> y  
3  
>>> z  
'name'

### More operations on list

* lst.append(x) # append the value x at last pos
* lst.extend(seq) # add a sequence to list(from end)
* lst.insert(index, x) # insert with index and value x
* lst.remove(x) # remove by value
* lst.pop(i) # providing index
* lst.pop() # remove from last
* lst.index(x) \* # get the value of index x
* lst.count(x) \* # no. of repeatition of value x in a list
* lst.sort() # sort the list, elements should be of same data type
* lst.sort(reverse=True) # sorts from descending-ascending
* sorted(lst) \* # sorts lst, but doesn’t saves it(temporary)
* lst.reverse() # reverses the list

*Note*: every operation which isn’t containing \* is modifying the list permanantly.

# Lecture 7:

## Mutabale and Immutable types:

* tuples and list types look very similar
* however, there is one major difference: Lists are *mutable*
  + Contents of list can be modified
* tuples and strings are *immutable*
  + Contents can’t be modified

>>> indoor = ['badminton', 'table-tennis', 'chess']  
  
>>> outdoor = ['football', 'cricket']  
>>> games = (indoor, outdoor)  
>>> type(games)  
<class, 'tuple'>  
>>> games  
(['badminton', 'table-tennis', 'chess'], ['football', 'cricket'])  
# let's take another example  
>>> lower = 'pants', 'shorts'  
>>> upper = 'shirt', 't-shirt'  
>>> type(lower)  
<class, 'tuple'>  
>>> dress = [lower, upper]  
# we don't need to cover lower/upper in `''` as they are already defined  
>>> type(dress)  
<class, 'list'>  
>>> dress  
[('pants', 'shorts'), ('shirt', 't-shirt')]  
  
# now, lets try some for mutable and immutable  
>>> games[0]  
['badminton', 'table-tennis', 'chess']  
>>> games[0] = card # or 'card'  
Traceback (most recent call last):  
 File "<input>", line 1, in <module>  
 games[0] = 'card'  
TypeError: 'tuple' object does not support item assignmen  
# so, can't change tuple, it's immutable  
  
# let's make another tuple, but listing explicitly all elements  
>>> games1 = (['badminton', 'table-tennis', 'chess'], ['football', 'cricket'], ['solitaire', 'hearts', 'freecell'])  
>>> games1[:1]  
(['badminton', 'table-tennis', 'chess'],)  
>>> games1[:2] == games  
True  
>>> indoor[2] = 'carrom'  
>>> games1[:2] == games  
False

Now, let’s resolve some questions raised above: \* Why did tuple games changed when we changed indoor list’s element? - tuple games contains reference of lists indoor and outdoor. - Thus, changing *outdoor* updated *games* as well. \* But why games1 didn’t changed? - Because it contains a different list(even though elements are same, we gave it explicitly) - Two lists can have the same content - Modifying one will not change the other. - So, games and games1 tuple elements had co-incidently same elements even though address of those elements were different

So, we can also change element of games1 tuple or should I say, elements inside the list of tuple games1

>>> games1[0][2] = 'carrom'  
>>> games1[:2] == games  
True