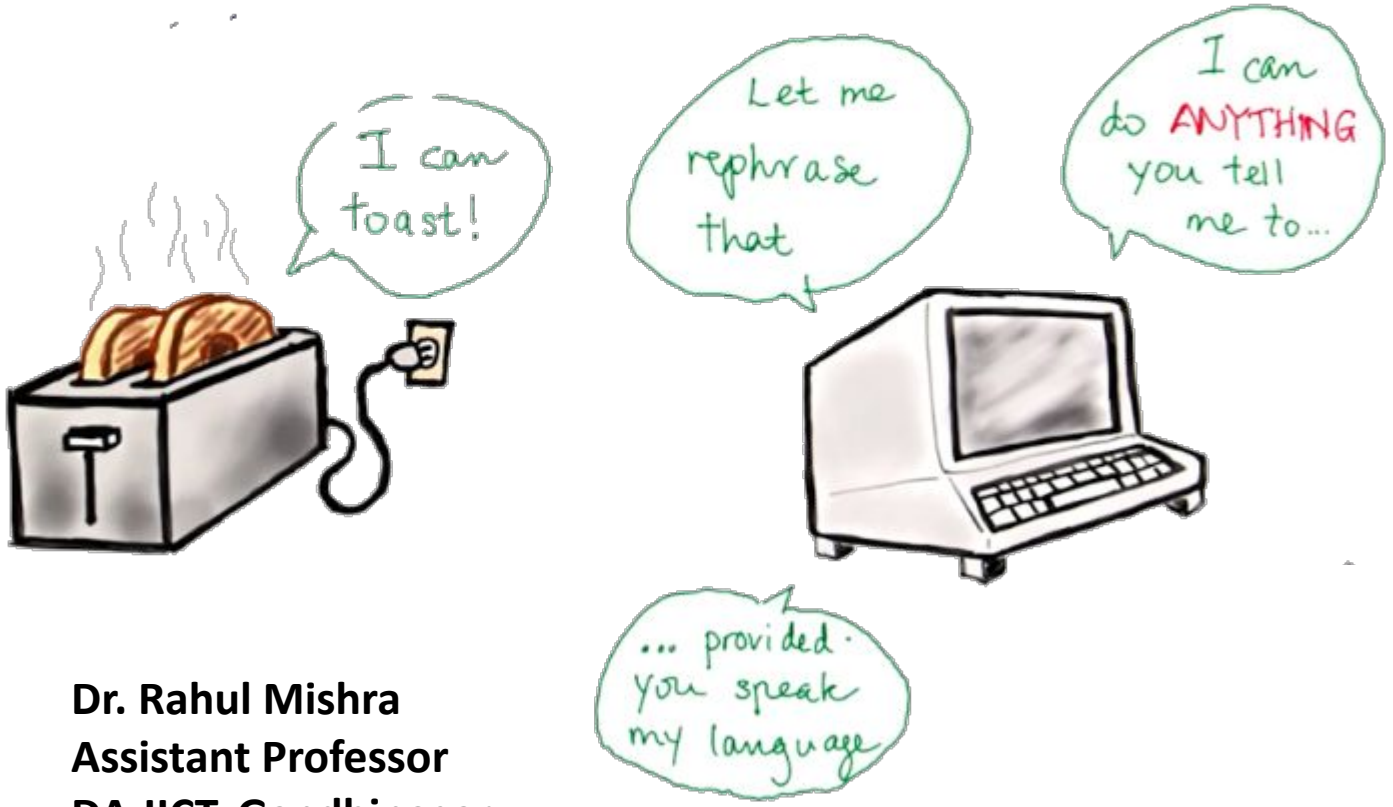
Programming Lab

Autumn Semester

Course code: PC503



Dr. Rahul Mishra
Assistant Professor
DA-IICT, Gandhinagar

Lecture 10

Different Data Types

List Comprehensions

- A list comprehension consists of brackets containing an expression followed by a for clause, then zero or more for or if clauses.
- The result will be a new list resulting from evaluating the expression in the context of the for and if clauses which follow it.
- For example, this listcomp combines the elements of two lists if they are not equal:that

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
>>>
```

List Comprehensions

```
>>> combs = []
>>> for x in [1,2,3]:
...     for y in [3,1,4]:
...         if x != y:
...             combs.append((x, y))
...
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
>>>
```

List Comprehensions

```
>>> vec = [-4, -2, 0, 2, 4]
>>> # create a new list with the values doubled
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filter the list to exclude negative numbers
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # apply a function to all the elements
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # call a method on each element
>>> freshfruit = ['  banana', '  loganberry ', 'passion fruit  ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> # create a list of 2-tuples like (number, square)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # the tuple must be parenthesized, otherwise an error is raised
>>> [x, x**2 for x in range(6)]
  File "<stdin>", line 1
    [x, x**2 for x in range(6)]
               ^
SyntaxError: invalid syntax
>>> # flatten a list using a listcomp with two 'for'
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
```

List Comprehensions

```
>>> from math import pi
>>> [str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

Nested List Comprehensions

```
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
>>> matrix
[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
```

```
>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
>>>
```

List Comprehensions

```
>>> transposed = []
>>> for i in range(4):
...     transposed.append([row[i] for row in matrix])
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

```
>>> transposed = []
>>> for i in range(4):
...     # the following 3 lines implement the nested
listcomp
...     transposed_row = []
...     for row in matrix:
...         transposed_row.append(row[i])
...     transposed.append(transposed_row)
...
 >>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

In the real world, you should prefer built-in functions to complex flow statements.
The zip() function would do a great job for this use case:

```
list(zip(*matrix))
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

The del statement

There is a way to remove an item from a list given its index instead of its value: the del statement.

This differs from the pop() method which returns a value.

The del statement can also be used to remove slices from a list or clear the entire list (which we did earlier by assignment of an empty list to the slice).

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
>>>
```

The del statement

del can also be used to delete entire variables:

```
del a
```

Tuples and Sequences

We saw that lists and strings have many common properties, such as indexing and slicing operations.

They are two examples of sequence data types (see Sequence Types — list, tuple, range).

Since Python is an evolving language, other sequence data types may be added.

There is also another standard sequence data type: the tuple.

The del statement

del can also be used to delete entire variables:

```
del a
```

Tuples and Sequences

We saw that lists and strings have many common properties, such as indexing and slicing operations.

They are two examples of sequence data types (see Sequence Types — list, tuple, range).

Since Python is an evolving language, other sequence data types may be added.

There is also another standard sequence data type: the tuple.

Tuples and Sequences

A tuple consists of a number of values separated by commas, for instance:

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
```

Tuples may be nested:

```
>>> u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

Tuples are immutable:

```
 >>>t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item
assignment
```

Tuples and Sequences

but they can contain mutable objects:

```
>>> v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
>>>
```

1. On output tuples are always enclosed in parentheses so that nested tuples are interpreted correctly;

2. They may be input with or without surrounding parentheses, although often parentheses are necessary anyway (if the tuple is part of a larger expression).

3. It is not possible to assign to the individual items of a tuple, however it is possible to create tuples that contain mutable objects, such as lists.

Tuples and Sequences

1. Though tuples may seem similar to lists, they are often used in **different situations and for different purposes.**
2. Tuples are immutable and usually contain a **heterogeneous sequence of elements** that are accessed via indexing.
3. Lists are mutable, and their elements are **usually homogeneous and are accessed by iterating over the list.**
4. A special problem is the construction of tuples containing **0 or 1** items:

Empty tuples are constructed by an empty pair of parentheses; a tuple with one item is constructed by following a value with a comma (it is not sufficient to enclose a single value in parentheses).

Tuples and Sequences

```
>>> empty = ()
>>> singleton = 'hello',    # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton = 'hello'
>>> len(singleton)
5
>>> singleton
'hello'
>>> singleton = 'hello',    # <-- note trailing comma
>>> len(singleton)
1
>>> singleton
('hello',)
>>>
```

Tuples and Sequences

The statement t = 12345, 54321, 'hello!' is an example of tuple packing: the values 12345, 54321 and 'hello!' are packed together in a tuple. The reverse operation is also possible:

```
>>> t = 12345, 54321, 'hello!'
>>> t
(12345, 54321, 'hello!')
>>>
>>> x, y, z = t
>>> t
(12345, 54321, 'hello!')
>>> x
12345
>>> y
54321
>>> z
'hello!'
>>>
```

Sets

1. A set is an unordered collection with no duplicate elements.

2. Basic uses include membership testing and eliminating duplicate entries.

3. Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.

Curly braces or the set() function can be used to create sets.

Note: to create an empty set you have to use set(), not {};

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)  #Shows the duplicate elements are removed
{'banana', 'apple', 'orange', 'pear'}
>>>
>>> 'orange' in basket                # fast membership testing
True
>>> 'crabgrass' in basket
False
>>>
```

Sets

Demonstrate set operations on unique letters from two words

```
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a
{'d', 'b', 'r', 'a', 'c'}
>>> a - b
{'d', 'b', 'r'}
>>> a | b # letters in a or b or both
{'m', 'd', 'b', 'r', 'z', 'a', 'c', 'l'}
>>> a & b
{'a', 'c'}
>>> a ^ b
{'m', 'r', 'd', 'b', 'l', 'z'}
>>>
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'d', 'r'}
>>>
```

Sets

Demonstrate set operations on unique letters from two words

```
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a
{'d', 'b', 'r', 'a', 'c'}
>>> a - b
{'d', 'b', 'r'}
>>> a | b # letters in a or b or both
{'m', 'd', 'b', 'r', 'z', 'a', 'c', 'l'}
>>> a & b
{'a', 'c'}
>>> a ^ b
{'m', 'r', 'd', 'b', 'l', 'z'}
>>>
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'d', 'r'}
>>>
```