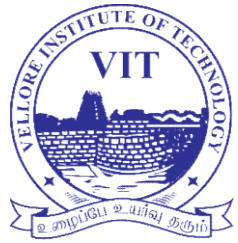
ITA6017
Python Programming

Dr. Arun Pandian J

Module 3: Pythons List, Tuples, Dictionaries & Sets



Lists and its operations, Ranges: Iterators and its purpose, Tuples: Operation and usage, Python Dictionaries, examples on Dictionaries, Sets and its operations

What is Not a “Collection”?

Most of our variables have one value in them - when we put a new value in the variable, the old value is overwritten

```
$ python
>>> x = 2
>>> x = 4
>>> print(x)
4
```

List
A List is a Kind of Collection

A collection allows us to put many values in a single “variable”

A collection is nice because we can carry all many values around in one convenient package.

friends = ['Joseph', 'Glenn', 'Sally']

carryon = ['socks', 'shirt', 'perfume']

List Constants

- List constants are surrounded by square brackets and the elements in the list are separated by commas
- A list element can be any Python object - even another list
- A list can be empty

```
>>> print([1, 24, 76])
[1, 24, 76]
>>> print(['red', 'yellow', 'blue'])
['red', 'yellow', 'blue']
>>> print(['red', 24, 98.6])
['red', 24, 98.6]
>>> print([ 1, [5, 6], 7])
[1, [5, 6], 7]
>>> print([])
[]
```

We Already Use Lists!


```
for i in [5, 4, 3, 2, 1] :
    print(i)
print('Blastoff!')
```

```
5
4
3
2
1
Blastoff!
```

Lists and Definite Loops - Best Pals

```
friends = ['Joseph', 'Glenn', 'Sally']
for friend in friends :
    print('Happy New Year:',  friend)
print('Done!')
```

```
Happy New Year: Joseph
Happy New Year: Glenn
Happy New Year: Sally
Done!
```

```
z = ['Joseph', 'Glenn', 'Sally']
for x in z:
    print('Happy New Year:',  x)
print('Done!')
```

Looking Inside Lists

Just like strings, we can get at any single element in a list using an index specified in square brackets

Joseph	Glenn	Sally
0	1	2

```
>>> friends = [ 'Joseph', 'Glenn', 'Sally' ]
>>> print(friends[1])
Glenn
>>>
```

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 = 'Banana'
>>> fruit[0] = 'b'
Traceback
TypeError: 'str' object does not
support item assignment
>>> x = fruit.lower()
>>> print(x)
banana
>>> lotto = [2, 14, 26, 41, 63]
>>> print(lotto)
[2, 14, 26, 41, 63]
>>> lotto[2] = 28
>>> print(lotto)
[2, 14, 28, 41, 63]
```

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 (such as a string...)

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

Using the range Function

- The range function returns a list of numbers that range from zero to one less than the parameter

- We can construct an index loop using for and an integer iterator

```
>>> print(range(4))
[0, 1, 2, 3]
>>> friends = ['Joseph', 'Glenn', 'Sally']
>>> print(len(friends))
3
>>> print(range(len(friends)))
[0, 1, 2]
>>>
```

A Tale of Two Loops...


```
friends = ['Joseph', 'Glenn', 'Sally']

for friend in friends :
    print('Happy New Year:',  friend)

for i in range(len(friends)) :
    friend = friends[i]
    print('Happy New Year:',  friend)
```

```
>>> friends = ['Joseph', 'Glenn', 'Sally']
>>> print(len(friends))
3
>>> print(range(len(friends)))
[0, 1, 2]
>>>
```

Happy New Year: Joseph

Happy New Year: Glenn

Happy New Year: Sally

Concatenating Lists Using +

We can create a new list by adding two existing lists together

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print(c)
[1, 2, 3, 4, 5, 6]
>>> print(a)
[1, 2, 3]
```

Lists Can Be Sliced Using :

```
>>> t = [9, 41, 12, 3, 74, 15]
>>> t[1:3]
[41,12]
>>> t[:4]
[9, 41, 12, 3]
>>> t[3:]
[3, 74, 15]
>>> t[:]
[9, 41, 12, 3, 74, 15]
```

Remember: Just like in strings, the second number is "up to but not including"

List Methods

```
>>> x = list()
>>> type(x)
<type 'list'>
>>> dir(x)
['append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
>>>
```

http://docs.python.org/tutorial/datastructures.html

Building a List from Scratch

- 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]
>>> stuff.append('cookie')
>>> print(stuff)
['book', 99, 'cookie']
```

Is Something in a List?

- Python provides two operators that let you check if an item is in a list
- These are logical operators that return True or False
- They do not modify the list

```
>>> some = [1, 9, 21, 10, 16]
>>> 9 in some
True
>>> 15 in some
False
>>> 20 not in some
True
>>>
```

Lists are in Order

A list can hold many items and keeps those items in the order until we do something to change the order

A list can be sorted (i.e., change its order)

The sort method (unlike in strings) means “sort yourself”

```
>>> friends = [ 'Joseph', 'Glenn', 'Sally' ]
>>> friends.sort()
>>> print(friends)
['Glenn', 'Joseph', 'Sally']
>>> print(friends[1])
Joseph
>>>
```

Built-in Functions and Lists

- There are a number of functions built into Python that take lists as parameters
- Remember the loops we built? These are much simpler.

```
>>> nums = [3, 41, 12, 9, 74, 15]
>>> print(len(nums))
6
>>> print(max(nums))
74
>>> print(min(nums))
3
>>> print(sum(nums))
154
>>> print(sum(nums)/len(nums))
25.6
```

```
total = 0
count = 0
while True :
    inp = input('Enter a number: ')
    if inp == 'done' : break
    value = float(inp)
    total = total + value
    count = count + 1

average = total / count
print('Average:', average)
```

```
Enter a number: 3
Enter a number: 9
Enter a number: 5
Enter a number: done
Average: 5.66666666667
```

```
numlist = list()
while True :
    inp = input('Enter a number: ')
    if inp == 'done' : break
    value = float(inp)
    numlist.append(value)

average = sum(numlist) / len(numlist)
print('Average:', average)
```

Task:

Consider a list (list = []). You can perform the following commands:

- Insert e at position i.
- Print the list.
- Delete the first occurrence of integer e.
- Insert integer e at the end of the list.
- Sort the list.
- Pop the last element from the list.
- Reverse the list.
- Find occurrence of integer e in the list.

Task:

- Given a list of ten numbers, the task is to write a Python program to find the second largest and second smallest number in the given list.

Task:

write a Python program to split the prime and composite number as separate list and order it.

Example:

Input:

[4, 2, 12, 3, 16, 5, 7, 64]

Output:

[2, 3, 5, 7]

[4, 12, 16, 64]

Tuples

Tuples Are Like Lists

Tuples are another kind of sequence that functions much like a list - they have elements which are indexed starting at 0

```
>>> x = ('Glenn', 'Sally', 'Joseph')
>>> print(x[2])
Joseph
>>> y = ( 1, 9, 2 )
>>> print(y)
(1, 9, 2)
>>> print(max(y))
9
```

```
>>> for iter in y:
...     print(iter)
...
1
9
2
>>>
```

but... Tuples are “immutable”

Unlike a list, once you create a tuple, you cannot alter its contents - similar to a string

```
>>> x = [9, 8, 7]
>>> x[2] = 6
>>> print(x)
>>>[9, 8, 6]
>>>
```

```
>>> y = 'ABC'
>>> y[2] = 'D'
Traceback:'str' object does
not support item
Assignment
>>>
```

```
>>> z = (5, 4, 3)
>>> z[2] = 0
Traceback:'tuple' object does
not support item
Assignment
>>>
```

Things not to do With Tuples

```
>>> x = (3, 2, 1)
>>> x.sort()
Traceback:
AttributeError: 'tuple' object has no attribute 'sort'
>>> x.append(5)
Traceback:
AttributeError: 'tuple' object has no attribute 'append'
>>> x.reverse()
Traceback:
AttributeError: 'tuple' object has no attribute 'reverse'
>>>
```

A Tale of Two Sequences


```
>>> l = list()
>>> dir(l)
['append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']

>>> t = tuple()
>>> dir(t)
['count', 'index']
```

Tuples are More Efficient

Since Python does not have to build tuple structures to be modifiable, they are simpler and more efficient in terms of memory use and performance than lists

So in our program when we are making “temporary variables” we prefer tuples over lists

Tuples and Assignment

We can also put a tuple on the left-hand side of an assignment statement

We can even omit the parentheses

```
>>> (x, y) = (4, 'fred')
>>> print(y)
fred
>>> (a, b) = (99, 98)
>>> print(a)
99
```

Tuples and Dictionaries

The items() method in dictionaries returns a list of (key, value) tuples

```
>>> d = dict()
>>> d['csev'] = 2
>>> d['cwen'] = 4
>>> for (k,v) in d.items():
...     print(k, v)
...
csev 2
cwen 4
>>> tups = d.items()
>>> print(tups)
dict_items([('csev', 2), ('cwen', 4)])
```

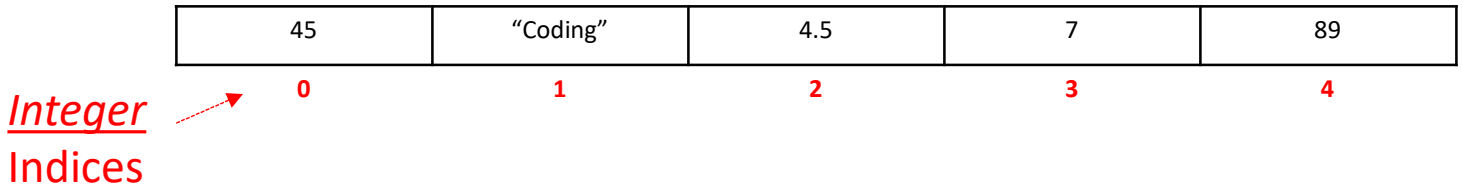
Tuples are Comparable

The comparison operators work with tuples and other sequences. If the first item is equal, Python goes on to the next element, and so on, until it finds elements that differ.

```
>>> (0, 1, 2) < (5, 1, 2)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
>>> ( 'Jones', 'Sally' ) < ('Jones', 'Sam')
True
>>> ( 'Jones', 'Sally') > ('Adams', 'Sam')
True
```

Dictionaries

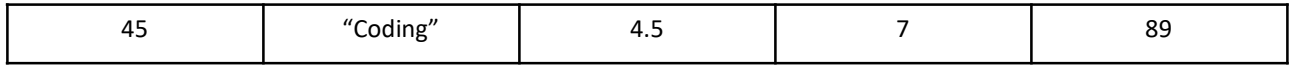
- Lists, tuples, and strings hold elements with *only integer* indices



- In essence, each element has an *index* (or a *key*) which can *only* be an integer, and a *value* which can be of any type (e.g., in the above list/tuple, the first element has key 0 and value 45)
  - *What if we want to store elements with non-integer indices (or keys)?*

Dictionaries

- In Python, you can use a dictionary to store elements with *keys of any hashable types* (e.g., integers, floats, Booleans, strings, and tuples; but not lists and dictionaries themselves) and *values of any types*



keys of different types

Values of different types

- The above dictionary can be defined in Python as follows:

```
dic = {"NUM":45, 1000:"coding", 2000:4.5, 3.4:7, "XXX":89}
```

key *value*

Each element is a *key*:*value* pair, and elements are separated by commas

Dictionaries

- To summarize, dictionaries:
  - Can contain any and different types of elements (i.e., hashable keys & values)
  - Can contain only *unique* keys but duplicate values

```
dic2 = {"a":1, "a":2, "b":2}
print(dic2)
```

Output: {'a': 2, 'b': 2}

The element “a”:2 will override the element “a”:1 because only ONE element can have key “a”

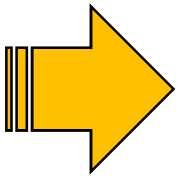
  - Can be indexed *but only* through keys (i.e., dic2[“a”] will return 1 but dic2[0] will return an error since there is no element with key 0 in dic2)

Dictionaries

- To summarize, dictionaries:
  - CANNOT be concatenated
  - Can be nested (e.g., d = {"first":{1:1}, "second":{2:"a"}}
  - Can be passed to a function and will result in a *pass-by-reference* and not *pass-by-value* behavior since they are *mutable* (similar to lists)

```
def func1(d):
    d["first"] = [1, 2, 3]

dic = {"first":{1:1},
"second":{2:"a"}}
print(dic)
func1(dic)
print(dic)
```



Output:

{'first': {1: 1}, 'second': {2: 'a'}}
{'first': [1, 2, 3], 'second': {2: 'a'}}

Dictionaries

- To summarize, dictionaries:
  - Can be iterated or looped over

```
dic = {"first": 1, "second": 2, "third": 3}
for i in dic:
    print(i)
```

Output:

first
second
third

ONLY the keys will be returned.

How to get the values?

Dictionaries

- To summarize, dictionaries:
  - Can be iterated or looped over

```
dic = {"first": 1, "second": 2, "third": 3}
for i in dic:
    print(dic[i])
```

Output:

1
2
3

Values can be accessed via indexing!

Adding Elements to a Dictionary

- How to add elements to a dictionary?
  - By indexing the dictionary via a key and assigning a corresponding value

```
dic = {"first": 1, "second": 2, "third": 3}
print(dic)
dic["fourth"] = 4
print(dic)
```

Output:

```
{'first': 1, 'second': 2, 'third': 3}
{'first': 1, 'second': 2, 'third': 3, 'fourth': 4}
```

Adding Elements to a Dictionary

- How to add elements to a dictionary?
  - By indexing the dictionary via a key and assigning a corresponding value

```
dic = {"first": 1, "second": 2, "third": 3}
print(dic)
dic[”second"] = 4
print(dic)
```

If the key already exists, the value will be overridden

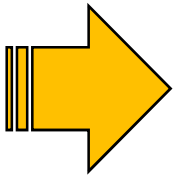
Output:

{'first': 1, 'second': 2, 'third': 3}
{'first': 1, 'second’: 4, 'third': 3}

Deleting Elements to a Dictionary

- How to delete elements in a dictionary?
  - By using **del**

```
dic = {"first": 1, "second": 2, "third": 3}
print(dic)
dic["fourth"] = 4
print(dic)
del dic["first"]
print(dic)
```



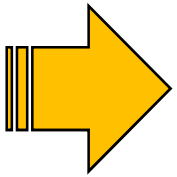
Output:

```
{'first': 1, 'second': 2, 'third': 3}
{'first': 1, 'second': 2, 'third': 3, 'fourth': 4}
{'second': 2, 'third': 3, 'fourth': 4}
```

Deleting Elements to a Dictionary

- How to delete elements in a dictionary?
  - Or by using the function **pop(key)**

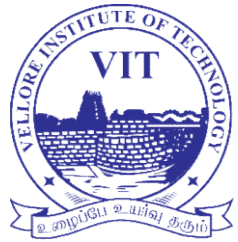
```
dic = {"first": 1, "second": 2, "third":
3}
print(dic)
dic["fourth"] = 4
print(dic)
dic.pop("first")
print(dic)
```



Output:

{'first': 1, 'second': 2, 'third': 3}
{'first': 1, 'second': 2, 'third': 3, 'fourth': 4}
{'second': 2, 'third': 3, 'fourth': 4}

Dictionary Functions



- Many other functions can also be used with dictionaries

Function	Description
dic.clear()	Removes all the elements from dictionary dic
dic.copy()	Returns a copy of dictionary dic
dic.items()	Returns a list containing a tuple for each key-value pair in dictionary dic
dic.get(k)	Returns the value of the specified key k from dictionary dic
dic.keys()	Returns a list containing all the keys of dictionary dic
dic.pop(k)	Removes the element with the specified key k from dictionary dic

Dictionary Functions

- Many other functions can also be used with dictionaries

Function	Description
dic.popitem()	Removes the last inserted key-value pair in dictionary dic
dic.values()	Returns a list of all the values in dictionary dic

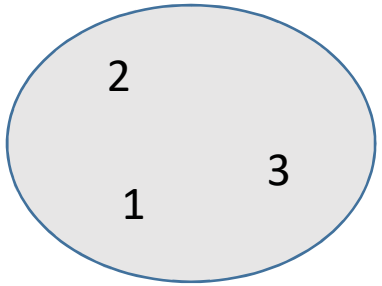
Sets

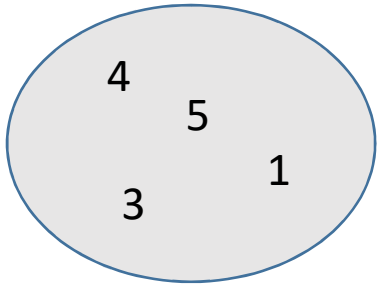
- Mathematical set: a collection of values, without duplicates or order
- Order does not matter

  { 1, 2, 3 } == { 3, 2, 1 }
- No duplicates

  { 3, 1, 4, 1, 5 } == { 5, 4, 3, 1 }
- For every data structure, ask:
  - How to create
  - How to query (look up) and perform other operations
    - (Can result in a new set, or in some other datatype)
  - How to modify

  Answer: http://docs.python.org/3/library/stdtypes.html#set





Two ways to create a set

1. Direct mathematical syntax:

```
odd = {1, 3, 5}
prime = {2, 3, 5}
```

Note: Cannot use "`{}`" to express empty set: it means something else ☹. Use `set()` instead.

2. Construct from a **list:** (also from a tuple or string)

```
odd = set([1, 3, 5])
prime = set([2, 3, 5])
empty = set([]) # or set()
```

Set operations

```
odd = {1, 3, 5}
prime = {2, 3, 5}
```

- membership ∈ Python: **`in`** **`4 in prime`** ⇒ False
- union ∪ Python: **`|`** **`odd | prime`** ⇒ {1, 2, 3, 5}
- intersection ∩ Python: **`&`** **`odd & prime`** ⇒ {3, 5}
- difference \ or - Python: **`-`** **`odd - prime`** ⇒ {1}

Think in terms of **set operations**,
not in terms of iteration and element operations

- Shorter, clearer, less error-prone, faster

Although we can do iteration over sets:

```
# iterates over items in arbitrary order
for item in myset:
    ...
```

But we *cannot* index into a set to access a specific element.

Practice with sets

```
z = {5, 6, 7, 8}
y = {1, 2, 3, 1, 5}
k = z & y
j = z | y
m = y - z
n = z - y
```

Modifying a set

- **Add** one element to a set:

```
myset.add(newelt)
myset = myset | {newelt}
```

- **Remove** one element from a set:

```
myset.remove(elt)    # elt must be in myset or raises error
myset.discard(elt)   # never errors
myset = myset - {elt}
```

What would this do?

```
myset = myset - elt
```

- Remove and return an arbitrary element from a set:

```
myset.pop()
```

Note: **add**, **remove** and **discard** all return **None**

Practice with sets


```
z = {5, 6, 7, 8}
y = {1, 2, 3, 1, 5}
p = z
q = set(z)  # Makes a copy of set z
z.add(9)
q = q | {35}
z.discard(7)
q = q - {6, 1, 8}
```

Aside: List vs. set operations (1)

Find the common elements **in both** list1 and list2:

```
out1 = []
for elem in list2:
    if elem in list1:
        out1.append(elem)
```

Find the common elements **in both** set1 and set2:

```
set1 & set2
```

Much shorter, clearer, easier to write with sets!

Aside: List vs. set operations(2)

Find elements in **either** list1 or list2 (**or both**) (without duplicates):

```
out2 = list(list1)  # make a copy
for elem in list2:
    if elem not in list1: # don't append elements already in out2
        out2.append(elem)
```

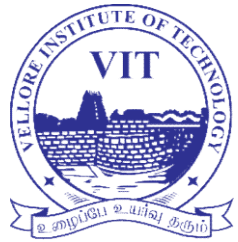
Another way:

```
out2 = list1 + list2  # if an item is in BOTH lists, it will appear TWICE!
for elem in out1:     # out1 = common elements in both lists
    out2.remove(elem) # Remove common elements, leaving just a single copy
```

Find the elements in **either** set1 or set2 (**or both**):

```
set1 | set2
```

Aside: List vs. set operations(3)



Find the elements in **either** list **but not in both**:

```
out3 = []
out2 = list1 + list2  # if an item is in BOTH lists, it will appear TWICE!
for elem in out2:
    if elem not in list1 or elem not in list2:
        out3.append(elem)
```

Find the elements in **either** set **but not in both**:

```
set1 - set2 | set2 - set1
set1 ^ set2
```

Not every value may be placed in a set

- Set *elements* must be **immutable** values
  - int, float, bool, string, *tuple*
  - *not*: list, set, dictionary
- The set itself is **mutable** (e.g. we can add and remove elements)

- **Aside:** *frozenset* must contain immutable values and is itself immutable (cannot add and remove elements)

Why not?

- Goal: only set operations change the set
  - after "**`myset.add(x)`**", **`x in myset`** ⇒ True
  - **`y in myset`** always evaluates to the same value

  Both conditions should hold until **`myset`** itself is changed
- Mutable elements can violate these goals

```
list1 = ["a", "b"]
list2 = list1
list3 = ["a", "b"]
myset = { list1 }
```
⇐ **Hypothetical; actually illegal in Python!**

`list1 in myset` ⇒ True

`list3 in myset` ⇒ True

`list2.append("c")` ⇐ not modifying **`myset`** "directly"

`list1 in myset` ⇒ ??? modifying **`myset`** "indirectly" would lead to different results

`list3 in myset` ⇒ ???

Thank You

Prepared by Dr. Arun Pandian J