SE-019 Advanced Programming

Tauseef Ahmed, PhD
Tauseef.Ahmed@eek.ee

Lecture 3: Python Built-In Data Structures

- List
- Tuple
- Dictionary
- Set

In Programming …

- Algorithm
  - A set of rules or steps used to solve a problem
- Data Structure
  - A particular way of organizing data in a computer

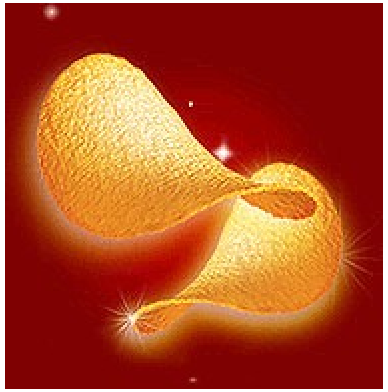
https://en.wikipedia.org/wiki/Algorithm
https://en.wikipedia.org/wiki/Data_structure

What is a Collection?

- A collection is nice because we can put more than one value in it and carry them all around in one convenient package
- We have a bunch of values in a single “variable”
- We do this by having more than one place “in” the variable
- We have ways of finding the different places in the variable
- There are **two** types of collections

Two Kinds of Collections..

- List
  - A linear collection of values that stay in order



- Dictionary
  - A “bag” of values (unordered), each with its own label

Lists

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' ]
```

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 = 2.14
>>> x = 4
>>> print(x)
4
```

Lists

- a **list** is a collection (or sequence) of values (position orderly)
- values in the list are called **elements** or **items**
  - in the list, elements can be of any type.
- Lists have no fixed size
- Lists are mutable (elements can be changed)
- [ ] is used to define list, e.g.
  - [ ] defines an empty list
  - [1, 2, 3, 5]
  - ['cat', 'bat', 'rat', 'elephant']
  - ['hello', 3.1415, True, None, 42]
- Python’s lists may be resonant of arrays in other languages, but they tend to be more powerful

https://docs.python.org/

- 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([]) #empty list
```

List Index

- Same as strings, **[ ]** operator is used to index the individual elements or items in the list.
  - City = [‘Tallinn’, ‘Parnu’],   City[0]   gives   >>> ‘Tallinn’

Negative index

….3   2   1

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

0   1   2   3 ….. .

Positive index

- Negative index counts from the left side
  - a[-3]   gives   >>> 4
- Any integer expression can be used as an index
- If you try to read or write an element that does not exist, you get an *IndexError*

Lists and Loops

```
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!

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]
```

List Operations

- Lists are sequences so they support sequence operations same as strings
- Lists can be nested (nested list is considered as 1 element
- **in, not in** operators also work for lists, e.g. >>> 3 in a (True)
- ***** repeats (multiplicate) the list, e.g. >>>list*2
- **+** concatenates lists, e.g. >>>list1 + list2

- <u>Slicing</u> a list returns a new list, >>> list1[1:3]
  - list1[:5] -> if first index is omitted, slice will start from beginning
  - list1[3:] -> if second index is omitted, slice will be till the end of list.

Slicing

alist[Initial : End : Step **]**

```
>>> 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: the second number is “up to but not including”

Is something in a list?

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

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

Concatenation or combining two lists: +

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’ Methods*

- list1.**append(**x**)**: adds an item to the end of the list
- list1.**extend(**list2**)**: takes a list as an argument and appends to the one called for.
- list1.**insert(**i, x**)**: insert the element x at given index i
- list1.**pop(**i**)**: to delete an element from the list at the index i, if no argument is provided then last item from the list is deleted.
- list1.**clear()**: removes all the elements (empty list).
- list1.**index(**x**)**: returns the index of the element x
- list1.**count(**x**)**: counts the occurrences of x
- list1.**sort()**: sorts the list
- list1.**reverse()**: reverse the elements of the list in place
- list1.**copy()**: returns the copy of list

*https://docs.python.org/3/tutorial/datastructures.html

List constructor

Built-in function

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

Lists and Built-in Functions

- **len(**list1**)**: length of the list in terms of items or elements
- **max(**list1**)**: maximum from the list1
- **min(**list1**)**: minimum from the list1
- **sum(**list1**)**: adds all the item in the list1 (only for numeric lists)
- **del** : remove slices from a list or clear the entire list. It can also remove the whole object from the memory

```
>>> 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
```

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 means “sort yourself”
- You can also pass True for the reverse keyword argument to have sort() sort the values in reverse order.

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

Lists and Strings

- string is a sequence of characters, where as list is a sequence of values
- string.**split():** can give us the words in list form, e.g.
  - *st* = 'this is test string’
  - *lis* = st.split() will give > ['this', 'is', 'test', 'string’]

- **join()** will join words from a list to make a string, delimiter can be use to insert string character of choice between list elements.
  - delimiter = ' '
  - delimiter.join(*lis*) will return the same string as *st*

```
>>> abc = 'With three words'
>>> stuff = abc.split()
>>> print(stuff)
['With', 'three', 'words']
>>> print(len(stuff))
3
>>> print(stuff[0])
With
```

```
>>> print(stuff)
['With', 'three', 'words']
>>> for w in stuff :
...     print(w)
...
With
Three
Words
>>>
```

Aliasing

- Reference: association of a variable with an object
- Alias: object with more than one name and more than one names (different names are called alias)

List	**Strings**
a = [1, 2, 3] b = [1, 2, 3] a **is** b >>> False Though the are identical lists but there are not same objects, so a and b are not same references, i.e. not aliasing. >>> b = a b is a >>> True (now its aliasing)	a = ‘banana’ b = ‘banana’ a **is** b >>> True Python has created same string object ‘banana’ and it has two reference a and b, i.e. aliasing

References

- variables “store” strings and integer values.
  - Technically, variables are storing references to the computer memory locations where the values are stored
  - In Immutable data types when changing the of variable is actually making it refer to a completely different value in memory.

```
>>> spam = 42
>>> cheese = spam
>>> spam = 100
>>> spam
100
>>> cheese
42
```

- lists do not work this way, because, lists are mutable.
  - Reference is saved in the variable

```
>>> spam = [0, 1, 2, 3, 4, 5]
>>> cheese = spam # The reference is being copied, not the
list.
>>> cheese[1] = 'Hello!' # This changes the list value.
>>> spam
[0, 'Hello!', 2, 3, 4, 5]
>>> cheese # The cheese variable refers to the same list.
[0, 'Hello!', 2, 3, 4, 5]
```

Although Python variables technically contain references to values, we often casually say that the variable contains the *value*.

Lists as Arrays

- Python multidimensional arrays >>> M = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

  Array operations (print, add, substract, etc.)

  - ```
    for i in range(len(M)):
          for j in range(len(M[0])):
                print (M[i][j], end = ' ')
          print('\n’)
    ```

  Matrix Multiplication

  - ```
    for i in range(len(x)):
          for j in range(len(y[0])):
                for k in range(len(y)):
                      result[i][j] += x[i][k] * y[k][j]
    ```

NumPy Arrays*

- NumPy is the fundamental package for scientific computing with Python
  - a powerful N-dimensional array object
  - tools for integrating C/C++ and Fortran code
  - useful linear algebra, Fourier transform, and random number capabilities
- *import **numpy** as np*
- *a = np.array([1, 2, 3, 4])* or *a = np.array([[1, 2],[3, 4]])*
- Basic operations: addition, multiplication
  - *b = a + a* or *a + 3*, *b = a * a* (individual elements multiplication), or *b = a * 5*
- Matrix multiplication
  - *c = a.dot(b)*

*http://www.scipy-lectures.org/intro/numpy/operations.html

Basic operations

```
>>> import numpy as np
>>> a = np.array([0,1,2,3,4,5])
>>> a
array([0, 1, 2, 3, 4, 5])
>>> a.ndim
1
>>> a.shape
(6,)
```

```
>>> b = a.reshape((3,2))
>>> b
array([[0, 1],
       [2, 3],
       [4, 5]])
>>> b.ndim
2
>>> b.shape
(3, 2)
```

```
>>> b[1][0]=77
>>> b
array([[ 0,  1],
       [77,  3],
       [ 4,  5]])
>>> a
array([ 0,  1, 77,  3,  4,  5])
```

```
>>> c = a.reshape((3,2)).copy()
>>> c
array([[ 0,  1],
       [77,  3],
       [ 4,  5]])
>>> c[0][0] = -99
>>> a
array([ 0,  1, 77,  3,  4,  5])
>>> c
array([[-99,   1],
       [ 77,   3],
       [  4,   5]])
```

```
>>> a*2
array([ 2,  4,  6,  8, 10])
>>> a**2
array([ 1,  4,  9, 16, 25])
Contrast that to ordinary Python lists:
>>> [1,2,3,4,5]*2
[1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
>>> [1,2,3,4,5]**2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for ** or pow(): 'list' and
'int'
```

List Comprehensions

- concise way to create lists.
- **[***expression* **for** *var* **in** *iterable***]**, where expression is any valid expression, var is a variable name, and iterable is any iterable Python object.
- compress a list-building *for* loop into a single short, readable line
  - For example, assume we want to create a list of squares

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

  - or, equivalently:

```
squares = [x**2 for x in range(10)]
```

  - which is more concise and readable.

- Sometimes list comprehensions can consist of multiple values rather than one. multiple *for* and/or *if* statements can be used. *if* statements can be used on values/variables and iterator.

```
>>> [(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)]
```

- equivalent to: (Note that the order of the *for* and *if* statements is the same)

```
>>> 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)]
```

- can contain complex expressions and nested functions

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

- initial expression in a list comprehension can be any arbitrary expression, including another list comprehension

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

- following list comprehension will transpose rows and columns:

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

- so this example is equivalent to:

```
>>> 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]]
```

```
>>> 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]]
```

Tuple

- A tuple is a sequence of values much like a *list*
- values stored in a tuple can be any type, and they are indexed by integers
- tuples are immutable where as lists are mutable.
- Tuples are also comparable and hashable
  - An object is hashable if it has a hash value which never changes during its lifetime
  - Hashability makes an object usable as a dictionary key and a set member, because these data structures use the hash value internally.
- t = ('a', 'b', 'c', 'd', 'e’)

() are not mandatory but are recommended

- to create a tuple with a single element, a comma is mandatory
- t = 1,
- type (t)
- <type 'tuple'>

Without the comma Python treats the value as other data types, e.g.;
t = ‘a’, is tuple
t = ‘a’ is string
t = 1, is tuple
t = 1 is integer

- Examples:
  - tuple 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
>>>
```

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

```
>>> 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
>>>
```

- `tuple()`
  - To construct a tuple with built-in function tuple()
- `t = tuple ()` -> creates empty tuple
- `t = tuple((1,2,3,))`
- `t = tuple(‘string’)`
- `[]` -> used to index the single element of the tuple or slice from the tuple (same as lists)
- Positive index start from 0 (left side) and negative index start from right (same as lists)
- `t = 12345, 54321, 'hello!'` -> can be seen as packing of data into a tuple
- `x, y, z = t` -> unpacking of data, i.e. three elements of tuple are assigned to individual variables. Each corresponding to the class based on the contents.
- The number of elements should be same as number of unpacking variables. If not then there will be `ValueError`

Tuple operations

- `len(t)` -> gives the number of elements in tuple
- `max(t)` -> gives the maximum element of the tuple
- `min(t)` -> gives the minimum element of the tuple
- `sum(t)` -> returns the summation of all the elements of tuple
- Concatenation: + ,e.g. `t3 = t1 + t2` combines two tuple
- Repetition: *, e.g. `t2 = t1 * 3` repeats the t1 tuple 3 times
- Tuple is an iterable object so can be used in iterable context, e.g. in loops, list comprehension
- Sorting:
  - Built-in function **`sorted()`** can be used as there is no sort method in tuple

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

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

- **`t.index(x)`** -> gives the index of element x
- **`t.count(x)`** -> occurrences of x in tuple t
- the rule about tuple immutability applies only to the top level of the tuple itself, not to its contents.

```
>>> T = (1, [2, 3], 4)
>>> T[1] = 'spam'   #It should fail, tuple does not support changing
Traceback (most recent call last):
TypeError: 'tuple' object does not support item assignment
>>> T[1][0] = 'spam' #changing inside the tuple, a mutable content
>>> T
(1, ['spam', 3], 4)
```

- Multiple values can be returned using tuples

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

Tuple - Lists

- Immutability of tuples provides some integrity—tuple will not be changed through another reference elsewhere in a program
- Tuples can also be used in places that lists cannot—for example, as dictionary keys
- 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
- As a rule of thumb, lists are the tool of choice for ordered collections that might need to change; tuples can handle the other cases of fixed associations

Comparing Tuples

- Comparison operators work with tuples and other sequences
- Element by element comparison is performed
  - If they are equal, it 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
```

Dictionary

Python Dictionaries

- Dictionaries are Python's most powerful data collection
- Dictionaries allow us to do fast database-like operations in Python
- Dictionaries have different names in different languages
  - Associative Arrays - Perl / PHP
  - Properties or Map or HashMap - Java
  - Property Bag - C# / .Net

Dictionaries

- Item names are more meaningful than item positions
- Dictionaries are useful and can replace manual creation of low level data structures
- Unordered collection of arbitrary items (objects). (key + value => item)
- Items are stored and fetched by key, instead of by positional offset(index)
- Variable-length, heterogeneous, and arbitrarily nestable
- Dictionaries are mutable however do not support the sequence operations that work on strings and lists
- Implemented as hash tables.

- Defining dictionaries:
  - `d1 = {}`
  - `d2 = {'key1': 'value1', 'key2': 'value2'}`
  - `d3 = dict(name='Bob', age=40)` using built-in function `dict()`
  - `d4 = {2: 4, 4: 16, 6: 36}`
  - `d5 = {'employee1': {'name': 'Bob', 'age': 40}}` -> nested dictionaries.
  - `d6 = dict( zip(keys_list, values_list))`
- Dictionary values are accessed by `[ ]` using keys as indexing.
  - `d1['key1']` results in 'value1'
  - If the key is not in the dictionary, `KeyError` exception will occur.
- **`in`** operator check if key is present in dictionary, it does not checks for value. However, **`in`** operator for values is possible.
  - `name in d3 >>> True`, as name is present in the d3 as key
  - `vals  = d3.values()`
  - `'Bob' in vals >>> True`, as Bob is present in values

- List uses *index* the entries based on the position in the list

- Dictionaries are like bags - no order

- So we *index* the things we put in the dictionary with a “lookup tag” called *key*

```
>>> purse = dict()
>>> purse['money'] = 12
>>> purse['candy'] = 3
>>> purse['tissues'] = 75
>>> print(purse)
{'money': 12, 'tissues': 75,
'candy': 3}
>>> print(purse['candy'])
3
>>> purse['candy'] = purse['candy']
+ 2
>>> print(purse)
{'money': 12, 'tissues': 75,
'candy': 5}
```

Dictionary Operations

- **`len(`**dictionary**`)`** > return the number of key-value pairs (items)
- **`D.keys()`** > gives all the keys of the dictionary D
- **`D.values()`** > gives all the values of the dictionary D
- **`D.items()`** > all key + values
- **`D.copy`**() > returns copy of the dictionary for which it was called.
- **`D.clear()`** > removes all items
- **`D.update(D2)`** > dictionaries are merged by keys.
- **`D.get(`**`key, default_value`**`)`** > return the value of the key, if key is not found then *default_value* is returned. If *default_value* is not supplied then default **None** is returned
- **`D.pop`**`(key, default_values`**`)`** > removes/returns the value of the key, if key is not present then *default_value* is returned. If *default_value* is not supplied then default **None** is returned

- **D.popitem()** > removes/return the item (key – value pair)
- **D.setdefault(**key, default_value**)** >return the value of the key, if key is not found, it is added in dictionary with *default_value* and it this values is is returned. If *default_value* is not supplied then key is assigned as default **None** and it is returned.
- D[key] = value > add or modify a key
- **list(**D.keys()**)** > changing dictionary enteries, i.e. keys to list (same goes for values also)
- **del** D[key] > deleting entry by key

Traversing Dictionaries

- Dictionary are iterable so can be used in as the sequence in a **for** statement, it traverses the keys of the dictionary
  - it goes through all of the keys in the dictionary and looks up the values

```
>>> counts = { 'chuck' : 1 , 'fred' : 42, 'jan': 100}
>>> for key in counts:
...     print(key, counts[key])
...
jan 100
chuck 1
fred 42
>>>
```

Traversing Dictionaries

- You can get a list of keys, values, or items (both) from a dictionary

```
>>> jjj = {'chuck' : 1 , 'fred' : 42, 'jan': 100}
>>> print(list(jjj))
['jan', 'chuck', 'fred']
>>> print(jjj.keys())
['jan', 'chuck', 'fred']
>>> print(jjj.values())
[100, 1, 42]
>>> print(jjj.items())
[('jan', 100), ('chuck', 1), ('fred', 42)]
>>>
```

Dictionaries and Lists

- Dictionaries are like lists except that they use keys instead of numbers to look up values

```
>>> lst = list()
>>> lst.append(21)
>>> lst.append(183)
>>> print(lst)
[21, 183]
>>> lst[0] = 23
>>> print(lst)
[23, 183]
```

```
>>> ddd = dict()
>>> ddd['age'] = 21
>>> ddd['course'] = 182
>>> print(ddd)
{'course': 182, 'age': 21}
>>> ddd['age'] = 23
>>> print(ddd)
{'course': 182, 'age': 23}
```

```
>>> lst = list()
>>> lst.append(21)
>>> lst.append(183)
>>> print(lst)
[21, 183]
>>> lst[0] = 23
>>> print(lst)
[23, 183]
```

List

Key	Value
[0]	21
[1]	183

```
>>> ddd = dict()
>>> ddd['age'] = 21
>>> ddd['course'] = 182
>>> print(ddd)
{'course': 182, 'age': 21}
>>> ddd['age'] = 23
>>> print(ddd)
{'course': 182, 'age': 23}
```

Dictionary

Key	Value
['course']	182
['age']	21

Dictionaries- Tuples

- Tuples are hashable and lists are not, so if a composite key is required to use in a dictionary, a tuple is used as key;
  - `T1 = (2, 3, 4)`
  - `T2 = (7, 8, 9)`
  - `Table = { T1 : 88, T2 : 155 }`
  - `Table [ T1 ]  >>> 88`
  - `Table [(2, 3, 4)] >>> 88`

Dictionaries- Tuples

- 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)])
```

Sorting

- We can take advantage of the ability to sort a list of tuples to get a sorted version of a dictionary
- First we sort the dictionary by the key using the items() method and sorted() function

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> t = sorted(d.items())
>>> t
[('a', 10), ('b', 1), ('c', 22)]
>>> for k, v in sorted(d.items()):
...     print(k, v)
...
a 10
b 1
c 22
```

Sort by values instead of key

- If we could construct a list of tuples of the form (value, key) we could sort by value
- We do this with a for loop that creates a list of tuples

```
>>> c = {'a':10, 'b':1, 'c':22}
>>> tmp = list()
>>> for k, v in c.items() :
...     tmp.append( (v, k) )
...
>>> print(tmp)
[(10, 'a'), (22, 'c'), (1, 'b')]
>>> tmp = sorted(tmp, reverse=True)
>>> print(tmp)
[(22, 'c'), (10, 'a'), (1, 'b')]
```

Nested Dictionaries

- Indexing by key is a search operation, so dictionaries can replace many data structures e.g. struct or records
- Dictionaries can represent structural information (nested dictionaries)

```
rec = {'name': 'Bob',
       'jobs': ['developer', 'manager'],
       'web':  'www.bobs.org/˜Bob',
       'home': {'state': 'Overworked', 'zip': 12345}}
```

  - To fetch components of nested objects, simply put together indexing operations:

```
>>> rec['name']
'Bob'
>>> rec['jobs']
['developer', 'manager']
>>> rec['jobs'][1]
'manager'
>>> rec['home']['zip']
12345
```

Dictionary Comprehensions

- D = {key: value ***for*** (key, value) ***in*** iterable if expression}
- Examples:

- `D = {x: x**3 for x in range(10)}`

  `{0: 0, 1: 1, 2: 8, 3: 27, 4: 64, 5: 125, 6: 216, 7: 343, 8: 512, 9: 729}`

- `D = {x: x**3 for x in range(10) if x**3 % 4 == 0}`

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

- `D = dict.fromkeys(['a', 'b', 'c'], 0)`

  `D = {k:0 for k in ['a', 'b', 'c']}`

  {'a': 0, 'b': 0, 'c': 0}

Sets

- A set is an unordered collection of items.
- Every element is unique (no duplicates)and immutable (which cannot be changed).
- Sets, itself they are mutable, elements can be added and removed
- Sets can be used to perform mathematical set operations like union, intersection, symmetric difference etc.
- Basic uses include membership testing and eliminating duplicate entries

- A set is created by placing all the items (elements) inside curly braces { }, separated by comma or by using the built-in function **set( ).**
- to create an empty set you have to use **set( )**, not { }; the latter creates an empty dictionary
- Sets can have any number of items and they may be of different types (integer, float, tuple, string etc.).
- Sets cannot have a mutable element, like list, set or dictionary, as its element, only hash-able elements are permitted
- Examples:

```
>>> my_set = {1, 2, 3}
>>> print(my_set)
{1, 2, 3}
```

```
>>> a = set('abracadabra')
>>> a
{'d', 'b', 'r', 'a', 'c'}
```

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)
{'orange', 'apple', 'banana', 'pear'}
```

```
>>> my_set = {1.0, "Hello", (1, 2, 3)}
>>> print(my_set)
{1.0, 'Hello', (1, 2, 3)}
```

```
>>> my_set = {1, 2, [3, 4]}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

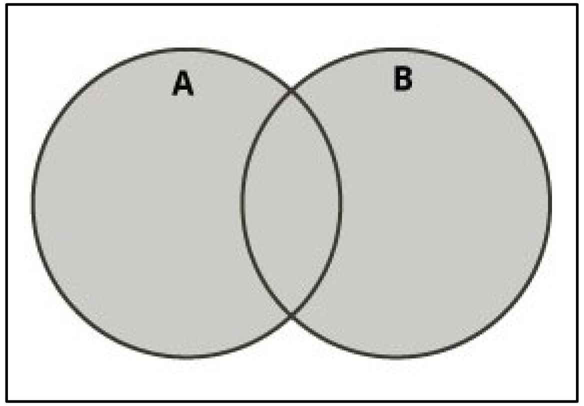
Sets Methods – Functions

- Fast membership testing with **`in`** or **`not in`** operator
- `s1.`**`add(`**`x`**`)`**: to add single element/item to the set s1
- `s1.`**`update(`**`s2`**`)`**: update multiple element to the set s1
- `s.`**`discard(`**`x`**`):`** remove the element x from the set.
- `s.`**`remove(`**`x`**`):`** remove the element x from the set. Give an error if element is not in set
- `s.`**`clear():`** removes all elements from the set.
- `s.`**`pop():`** can remove element from the set but since sets are unordered so pop will remove any arbitrary element.
- **`len(`**`s`**`):`** total elements in the set
- **`sorted(`**`s`**`):`** sort the elements in increasing order
- **`sum(`**`s`**`):`** returns the sum of the elements of set s

Sets Operations

- **Set Union**:
  - Union of set A and set B is a set of all elements from both sets.
  - Operator: **|**
  - Method: **`union()`**
  - Example:

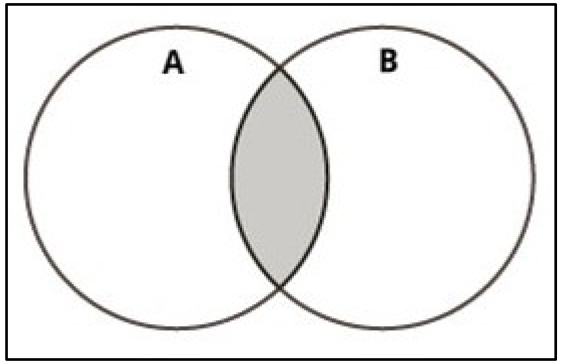
```
>>> A = {1, 2, 3, 4, 5}
>>> B = {4, 5, 6, 7, 8}
>>> print(A | B)
{1, 2, 3, 4, 5, 6, 7, 8}
>>> A.union(B)
{1, 2, 3, 4, 5, 6, 7, 8}
>>> B.union(A)
{1, 2, 3, 4, 5, 6, 7, 8}
```



Sets Operations

- **Set Intersection**:
  - Intersection of set A and set B is a set of elements that are common in both sets.
  - Operator: **&**
  - Method: **`intersection()`**
  - Example:

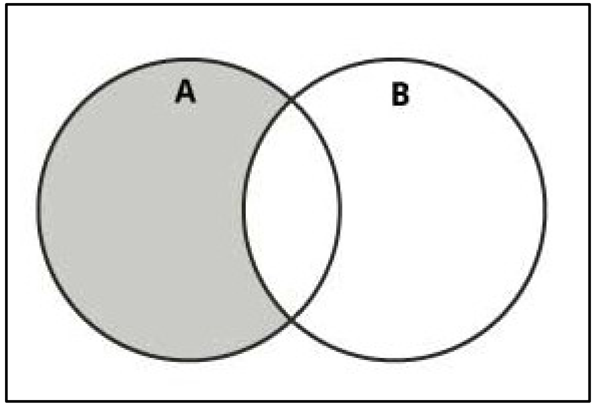
```
>>> A = {1, 2, 3, 4, 5}
>>> B = {4, 5, 6, 7, 8}
>>> print(A & B)
{4, 5}
>>> A.intersection(B)
{4, 5}
>>> B.intersection(A)
{4, 5}
```



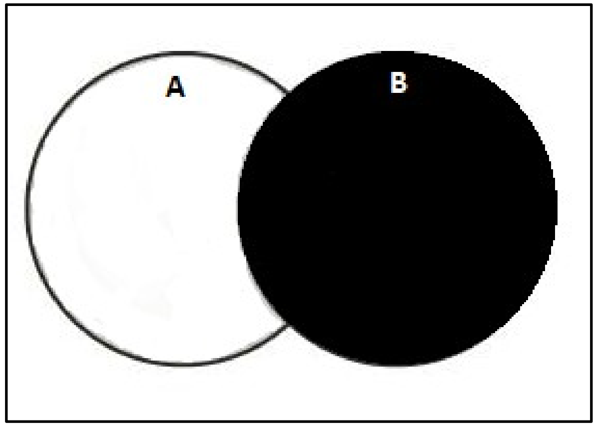
Sets Operations

- **Set Difference**:
  - A - B is a set of elements that are only in A but not in B.
  - B - A is a set of elements in B but not in A.
  - Operator: **`-`**
  - Method: **`difference()`**
  - Example:

```
>>> A = {1, 2, 3, 4, 5}
>>> B = {4, 5, 6, 7, 8}
>>> print(A - B)
{1, 2, 3}
>>> A.difference(B)
{1, 2, 3}
>>> print(B - A)
{8, 6, 7}
>>> B.difference(A)
{8, 6, 7}
```



A - B

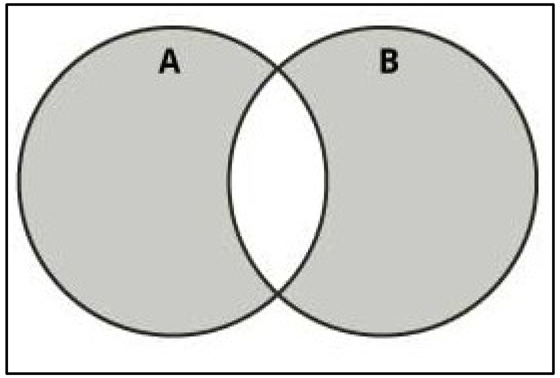


B - A

Sets Operations

- **Set Symetric Difference**:
  - Symmetric Difference of sets A and B is a set of elements in both A and B except those that are common in both.
  - Operator: **^**
  - Method: **`symmetric_difference()`**
  - Example:

```
>>> A = {1, 2, 3, 4, 5}
>>> B = {4, 5, 6, 7, 8}
>>> print(A ^ B)
{1, 2, 3, 6, 7, 8}
>>> A.symmetric_difference(B)
{1, 2, 3, 6, 7, 8}
>>> B.symmetric_difference(A)
{1, 2, 3, 6, 7, 8}
```



Set Comprehensions

- Same as lists comprehensions, sets also support comprehensions.
- Example:
- Set of all odd numbers raised to power 4, from 0-20
- `{pow(x, 4) for x in range(20) if x % 2 != 0}`
- `{x for x in 'abracadabra' if x not in 'abc'}`

Frozenset

- Frozenset is a new class that has the characteristics of a set, but its elements cannot be changed once assigned
- Frozensets are immutable sets(same as tuples)
- Methods: `copy()`, `difference()`, `intersection()`, `isdisjoint()`, `issubset()`, `issuperset()`, `symmetric_difference()`, `union()`.
- Since immutable so no add, remove methods.
- Example:
  - **`A = frozenset([1, 2, 3, 4])`**

Ranges

- The **range** type represents an immutable sequence of numbers and is commonly used for looping a specific number of times in **for** loops
- **range** is an iterable that generates items on demand
- **range(**`stop`**)**
- **range(**`start, stop[, step]`**)**
- The arguments, i.e. `start`, `stop` and `step` to the range constructor must be integers
- `step` argument is optional and if it is omitted, it defaults to 1
- If the `start` argument is omitted, it defaults to 0. If `step` is 0, `ValueError` is raised.
- For a positive step, the contents of a range *r* are determined by the formula:

  `r[i] = start + step * i` where `i >= 0` and r[i] < stop
- For a negative `step`, the contents of the range are still determined by the formula **`r[i] = start + step * i`** , but the constraints are `i >= 0` and `r[i] > stop`

- Ranges support negative indices, (same as other sequences) these are interpreted as indexing from the end of the sequence determined by the positive indices.
- Ranges implement all of the common sequence operations except concatenation and repetition
  - range objects can only represent sequences that follow a strict pattern and repetition and concentenation will usually violate that pattern
- `start`
  - The value of the start parameter (or 0 if the parameter was not supplied)
- `stop`
  - The value of the stop parameter
- `step`
  - The value of the step parameter (or 1 if the parameter was not supplied)

Examples

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(range(0, 30, 5))
[0, 5, 10, 15, 20, 25]
>>> list(range(0, 10, 3))
[0, 3, 6, 9]
>>> list(range(0, -10, -1))
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> list(range(0))
[]
>>> list(range(1, 0))
[]
```

Ranges vs (Lists/Tuples)

- The advantage of the range type over a regular list or tuple (sequences) is that a range object will always take the same (small) amount of memory, no matter the size of the range it represents
- Range stores the `start`, `stop` and `step` values, calculating individual items and subranges as needed
- association, element index lookup, slicing and support for negative indices (same as lists or tuples because range objects are sequences)
- range vs slices
  - slicing makes a copy of the string in both 2.X and 3.X, while range in 3.X do not create a list and for very large sequences, they may save memory in case of processing long sequences.

Examples

```
>>> r = range(0, 20, 2)
>>> r
range(0, 20, 2)
>>> 11 in r
False
>>> 10 in r
True
>>> r.index(10)
5
>>> r[5]
10
>>> r[:5]
range(0, 10, 2)
>>> r[-1]
18
```

Comparison

- Testing range objects for equality with == and `!=` compares them as sequences
- Two range objects are considered equal if they represent the same sequence of values.
  - Two range objects that compare equal might have different *start*, *stop* and *step* attributes
  - Example:
    - `range(0) == range(2, 1, 3)`
    - `range(0, 3, 2) == range(0, 4, 2)`