Python Programming

By

Subhadip Mukherjee

Dept. of Computer Science and BCA,
Kharagpur College,
Kharagpur, India

LECTURE 5

For CBCS 4th Semester

WHEN TO USE SETS

- When the elements must be unique.
- When you need to be able to modify or add to the collection.
- When you need support for mathematical set operations.
- When you don't need to store nested lists, sets, or dictionaries as elements.

CREATING SETS

- Create an empty set with the set constructor.

```
myset = set()
myset2 = set([]) # both are empty sets
```

- Create an initialized set with the set constructor or the { } notation. Do not use empty curly braces to create an empty set – you'll get an empty dictionary instead.

```
myset = set(sequence)
myset2 = {expression for variable in sequence}
```

HASHABLE ITEMS

The way a set detects non-unique elements is by indexing the data in memory, creating a hash for each element. This means that all elements in a set must be *hashable*.

All of Python's immutable built-in objects are hashable, while no mutable containers (such as lists or dictionaries) are. Objects which are instances of user-defined classes are also hashable by default.

MUTABLE OPERATIONS

The following operations are not available for frozensets.

- The add(x) method will add element x to the set if it's not already there. The remove(x) and discard(x) methods will remove x from the set.
- The pop() method will remove and return an arbitrary element from the set. Raises an error if the set is empty.
- The clear() method removes all elements from the set.

```
>>> myset = {x for x in 'abracadabra'}
>>> myset
set(['a', 'b', 'r', 'c', 'd'])
>>> myset.add('y')
>>> myset
set(['a', 'b', 'r', 'c', 'd', 'y'])
>>> myset.remove('a')
>>> myset
set(['b', 'r', 'c', 'd', 'y'])
>>> myset.pop()
'b'
>>> myset
set(['r', 'c', 'd', 'y'])
```

MUTABLE OPERATIONS CONTINUED

set |= *other* | ...

Update the set, adding elements from all others.

set &= *other* & ...

Update the set, keeping only elements found in it and all others.

set -= *other* | ...

Update the set, removing elements found in others.

set ^= *other*

Update the set, keeping only elements found in either set, but not in both.

MUTABLE OPERATIONS CONTINUED

```
>>> s1 = set('abracadabra')
>>> s2 = set('alacazam')
>>> s1
set(['a', 'b', 'r', 'c', 'd'])
>>> s2
set(['a', 'l', 'c', 'z', 'm'])
>>> s1 |= s2
>>> s1
set(['a', 'b', 'r', 'c', 'd', 'l', 'z', 'm'])
>>> s1 = set('abracadabra')
>>> s1 &= s2
>>> s1
set(['a', 'c'])
```

SET OPERATIONS

- The following operations are available for both set and frozenset types.
- Comparison operators >=, <= test whether a set is a superset or subset, respectively, of some other set. The > and < operators check for proper supersets/subsets.

```
>>> s1 = set('abracadabra')
>>> s2 = set('bard')
>>> s1 >= s2
True
>>> s1 > s2
True
>>> s1 <= s2
False
```

SET OPERATIONS

- Union: `set | other | ...`
  - Return a new set with elements from the set and all others.
- Intersection: `set & other & ...`
  - Return a new set with elements common to the set and all others.
- Difference: `set - other - ...`
  - Return a new set with elements in the set that are not in the others.
- Symmetric Difference: `set ^ other`
  - Return a new set with elements in either the set or other but not both.

SET OPERATIONS

```
>>> s1 = set('abracadabra')
>>> s1
set(['a', 'b', 'r', 'c', 'd'])
>>> s2 = set('alacazam')
>>> s2
set(['a', 'l', 'c', 'z', 'm'])
>>> s1 | s2
set(['a', 'b', 'r', 'c', 'd', 'l', 'z', 'm'])
>>> s1 & s2
set(['a', 'c'])
>>> s1 - s2
set(['b', 'r', 'd'])
>>> s1 ^ s2
set(['b', 'r', 'd', 'l', 'z', 'm'])
```

OTHER OPERATIONS

- `s.copy()` returns a shallow copy of the set *s*.
- `s.isdisjoint(other)` returns True if set *s* has no elements in common with set *other*.
- `s.issubset(other)` returns True if set *s* is a subset of set *other*.
- `len`, `in`, and `not in` are also supported.

WHEN TO USE TUPLES

- When storing elements that will not need to be changed.
- When performance is a concern.
- When you want to store your data in logical immutable pairs, triples, etc.

CONSTRUCTING TUPLES

- An empty tuple can be created with an empty set of parentheses.
- Pass a sequence type object into the tuple() constructor.
- Tuples can be initialized by listing comma-separated values. These do not need to be in parentheses but they can be.
- One quirk: to initialize a tuple with a single value, use a trailing comma.

```
>>> t1 = (1, 2, 3, 4)
>>> t2 = "a", "b", "c", "d"
>>> t3 = ()
>>> t4 = ("red", )
```

TUPLE OPERATIONS

Tuples are very similar to lists and support a lot of the same operations.

- Accessing elements: use bracket notation (e.g. `t1[2]`) and slicing.
- Use `len(t1)` to obtain the length of a tuple.
- The universal immutable sequence type operations are all supported by tuples.
  - +, *
  - in, not in
  - min(t), max(t), t.index(x), t.count(x)

PACKING/UNPACKING

Tuple packing is used to “pack” a collection of items into a tuple. We can unpack a tuple using Python’s multiple assignment feature.

```
>>> s = ("Susan", 19, "CS") # tuple packing
>>> (name, age, major) = s # tuple unpacking
>>> name
'Susan'
>>> age
19
>>> major
'CS'
```

WHEN TO USE DICTIONARIES

- When you need to create associations in the form of key:value pairs.
- When you need fast lookup for your data, based on a custom key.
- When you need to modify or add to your key:value pairs.

CONSTRUCTING A DICTIONARY

- Create an empty dictionary with empty curly braces or the dict() constructor.
- You can initialize a dictionary by specifying each key:value pair within the curly braces.
- Note that keys must be *hashable* objects.

```
>>> d1 = {}
>>> d2 = dict() # both empty
>>> d3 = {"Name": "Susan", "Age": 19, "Major": "CS"}
>>> d4 = dict(Name="Susan", Age=19, Major="CS")
>>> d5 = dict(zip(['Name', 'Age', 'Major'], ["Susan", 19, "CS"]))
>>> d6 = dict([('Age', 19), ('Name', "Susan"), ('Major', "CS")])
```

Note: zip takes two equal-length collections and merges their corresponding elements into tuples.

ACCESSING THE DICTIONARY

To access a dictionary, simply index the dictionary by the key to obtain the value. An exception will be raised if the key is not in the dictionary.

```
>>> d1 = {'Age':19, 'Name':"Susan", 'Major':"CS"}
>>> d1['Age']
19
>>> d1['Name']
'Susan'
```

UPDATING A DICTIONARY

Simply access a key:value pair to modify it or add a new pair. The del keyword can be used to delete a single key:value pair or the whole dictionary. The clear() method will clear the contents of the dictionary.

```
>>> d1 = {'Age':19, 'Name':"Susan", 'Major':"CS"}
>>> d1['Age'] = 21
>>> d1['Year'] = "Junior"
>>> d1
{'Age': 21, 'Name': 'Susan', 'Major': 'CS', 'Year': 'Junior'}
>>> del d1['Major']
>>> d1
{'Age': 21, 'Name': 'Susan', 'Year': 'Junior'}
>>> d1.clear()
>>> d1
{}
```

BUILT-IN DICTIONARY METHODS

```
>>> d1 = {'Age':19, 'Name':"Susan", 'Major':"CS"}
>>> d1.has_key('Age') # True if key exists
True
>>> d1.has_key('Year') # False otherwise
False
>>> d1.keys() # Return a list of keys
['Age', 'Name', 'Major']
>>> d1.items() # Return a list of key:value pairs
[('Age', 19), ('Name', 'Susan'), ('Major', 'CS')]
>>> d1.values() # Returns a list of values
[19, 'Susan', 'CS']
```

Note: `in`, `not in`, `pop(key)`, and `popitem()` are also supported.

ORDERED DICTIONARY

Dictionaries do not remember the order in which keys were inserted. An ordered dictionary implementation is available in the collections module. The methods of a regular dictionary are all supported by the OrderedDict class.

An additional method supported by OrderedDict is the following:

```
OrderedDict.popitem(last=True)   # pops items in LIFO order
```

ORDERED DICTIONARY

```
>>> # regular unsorted dictionary
>>> d = {'banana': 3, 'apple': 4, 'pear': 1, 'orange': 2}

>>> # dictionary sorted by key
>>> OrderedDict(sorted(d.items(), key=lambda t: t[0]))
OrderedDict([('apple', 4), ('banana', 3), ('orange', 2), ('pear', 1)])

>>> # dictionary sorted by value
>>> OrderedDict(sorted(d.items(), key=lambda t: t[1]))
OrderedDict([('pear', 1), ('orange', 2), ('banana', 3), ('apple', 4)])

>>> # dictionary sorted by length of the key string
>>> OrderedDict(sorted(d.items(), key=lambda t: len(t[0]))) OrderedDict([('pear',
1), ('apple', 4), ('orange', 2), ('banana', 3)])
```

Thank You