ADVANCED PYTHON PROGRAMMING

Data Collections

Peter Lo

Mutable Object

Mutable vs Immutable Objects in Python

- **Mutable object** means an object that can be changed after creating it (Using *same ID*).
  - Container-like types are probably mutable
- **Immutable object** means an object that can not be changed once you create it (*Or using another ID*).
  - Primitive-like types are probably immutable.

Class	Description	Immutable?
bool	Boolean value	✓
int	integer (arbitrary magnitude)	✓
float	floating-point number	✓
list	mutable sequence of objects	
tuple	immutable sequence of objects	✓
str	character string	✓
set	unordered set of distinct objects	
frozenset	immutable form of set class	✓
dict	associative mapping (aka dictionary)	

Example for Immutable Objects

- The original ID for x is 1681353920 when assigning value 5. When x is reassigned to 6, the original ID for x is delete, and a new ID for x is created.

```
In [3]: x = 5

In [4]: id(x)
Out[4]: 1681353920

In [5]: x = 6

In [6]: id(x)
Out[6]: 1681353952
```

List

List

- List is a collection of arbitrary objects, somewhat akin to an array in many other programming languages but more flexible.
- Lists are defined in Python by enclosing a comma-separated sequence of objects in square brackets [ ].

Python

```
>> a = ['foo', 'bar', 'baz', 'qux']

>>> print(a)
['foo', 'bar', 'baz', 'qux']
```

Note: Python does not have built-in support for Arrays, but Python lists can be used instead

Characteristics of List

1. Lists are ordered
2. Lists can contain any arbitrary objects
3. List elements can be accessed by index
4. Lists can be nested to arbitrary depth
5. Lists are mutable
6. Lists are dynamic

1. Lists are Ordered

- A list is not merely a collection of objects. It is an ordered collection of objects.
- The order in which you specify the elements when you define a list is an innate characteristic of that list and is maintained for that list's lifetime.

Python

```
>>> a = ['foo', 'bar', 'baz', 'qux']
>>> b = ['baz', 'qux', 'bar', 'foo']
>>> a == b
False
```

2. Lists can Contain Arbitrary Objects

- The elements of a list can be the same type
- A list can also contain any assortment of objects.

Python

```python
>>> a = [2, 4, 6, 8]
>>> a
[2, 4, 6, 8]
```

Python

```python
>>> a = [21.42, 'foobar', 3, 4, 'bark', False, 3.14159]
>>> a
[21.42, 'foobar', 3, 4, 'bark', False, 3.14159]
```

3. List Elements can be Accessed by Index

- Individual elements in a list can be accessed using an index in square brackets.
- This is exactly analogous to accessing individual characters in a string.
- List indexing is zero-based as it is with strings.

Python

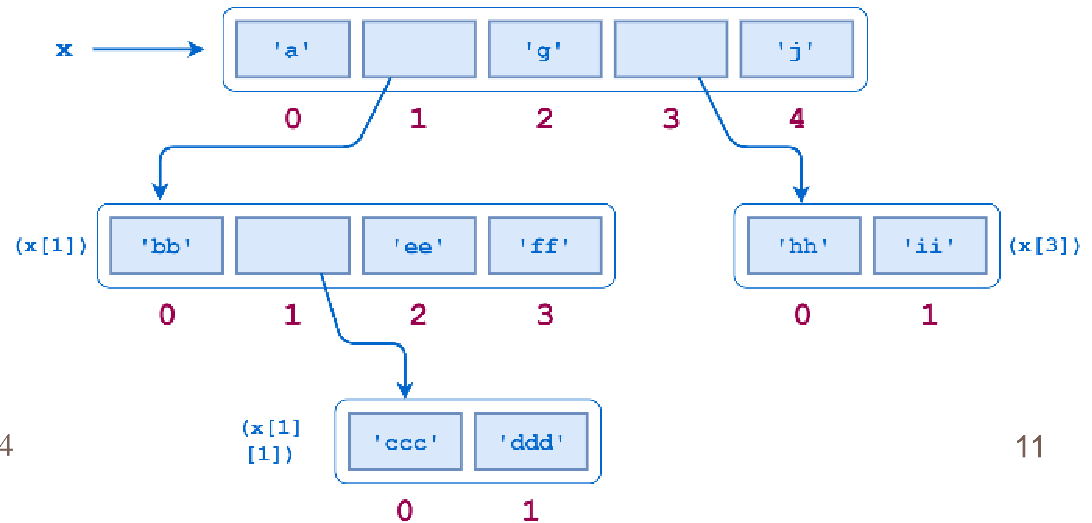
```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> a[0]
'foo'
>>> a[2]
'baz'
>>> a[5]
'corge'
```

4. Lists can be Nested

- An element in a list can be any sort of object that includes another list.
- A list can contain sublists, which in turn can contain sublists themselves, and so on to arbitrary depth.

Python

```
>>> x = ['a', ['bb', ['ccc', 'ddd'], 'ee', 'ff'], 'g', ['hh', 'ii'], 'j']
>>> x
['a', ['bb', ['ccc', 'ddd'], 'ee', 'ff'], 'g', ['hh', 'ii'], 'j']
```



5. Lists are Mutable

- Once a list has been created, elements can be added, deleted, shifted, and moved around at will.
- Python provides a wide range of ways to modify lists.

Method	Description
append(<obj>)	Appends an object to a list.
extend(<*iterable*>)	Extends a list with the objects from an iterable
insert(<*index*>, <*obj*>)	Inserts an object into a list
remove(<*obj*>)	Removes an object from a list
pop(<*index*>)	Removes an element from a list

6. Lists are Dynamic

- When items are added to a list, it grows as needed

Python

```python
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> a[2:2] = [1, 2, 3]
>>> a += [3.14159]
>>> a
['foo', 'bar', 1, 2, 3, 'baz', 'qux', 'quux', 'corge', 3.14159]
```

Python

```python
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> a[2:3] = []
>>> del a[0]
>>> a
['bar', 'qux', 'quux', 'corge']
```

Manipulating List by Built-in Function

- Several Python operators and built-in functions can also be used with lists in ways that are analogous to strings:
- The ***in*** and ***not in*** operators:

Python

```python
>>> a
['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> 'qux' in a
True
>>> 'thud' not in a
True
```

Manipulating List by Built-in Function (Cont.)

- The concatenation + and replication * operators:

Python

```
>>> a
['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> a + ['grault', 'garply']
['foo', 'bar', 'baz', 'qux', 'quux', 'corge', 'grault', 'garply']
>>> a * 2
['foo', 'bar', 'baz', 'qux', 'quux', 'corge', 'foo', 'bar', 'baz',
'qux', 'quux', 'corge']
```

Manipulating List by Built-in Function (Cont.)

- The ***len( )***, ***min( )***, and ***max( )*** functions:

Python

```
>>> a
['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> len(a)
6
>>> min(a)
'bar'
>>> max(a)
'qux'
```

Append to List

- ***append(***<*obj*>***)*** appends object <*obj*> to the end of list.

Python

```
>>> a = ['a', 'b']
>>> a.append(123)
>>> a
['a', 'b', 123]
```

Python

```
>>> a = ['a', 'b']
>>> x = a.append(123)
>>> print(x)
None
>>> a
['a', 'b', 123]
```

List methods modify the target list in place, they do not return a new list

Different between “append” and “+”

- When using **+** operator to concatenate a list, if the target operand is an iterable, then its elements are broken out and appended to the list individually

Python

```
>>> a = ['a', 'b']
>>> a + [1, 2, 3]
['a', 'b', 1, 2, 3]
```

- If an iterable is appended to a list with **.append( )**, it is added as a single object

Python

```
>>> a = ['a', 'b']
>>> a.append([1, 2, 3])
>>> a
['a', 'b', [1, 2, 3]]
```

Extend a List

- **extend()** adds the object to the end of a list, but the argument is expected to be an iterable. The items are added individually.

Python

```
>>> a = ['a', 'b']
>>> a.extend([1, 2, 3])
>>> a
['a', 'b', 1, 2, 3]
```

Python

```
>>> a = ['a', 'b']
>>> a += [1, 2, 3]
>>> a
['a', 'b', 1, 2, 3]
```

extend() behaves like the + operator. Since it modifies the list in place, it behaves like the += operator

Insert an Object to List

- **insert(**<*index*>, <*obj*>**)** inserts object <*obj*> into a list at the specified <*index*>.

Python

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> a.insert(3, 3.14159)
>>> a[3]
3.14159
>>> a
['foo', 'bar', 'baz', 3.14159, 'qux', 'quux', 'corge']
```

Remove an Object from List

- **remove(**<*obj*>**)** removes object <*obj*> from a list. If <*obj*> isn't in list, an exception is raised:

Python

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> a.remove('baz')
>>> a
['foo', 'bar', 'qux', 'quux', 'corge']

>>> a.remove('Bark!')
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    a.remove('Bark!')
ValueError: list.remove(x): x not in list
```

Remove an Object from List (Cont.)

- **pop(**<*index*>**)** removes an element from a list.
- This method differs from **remove()** in two ways:
  - You specify the index of the item to remove, rather than the object itself.
  - The method returns a value: the item that was removed.

Python

```python
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> a.pop(1)
'bar'
>>> a
['foo', 'baz', 'qux', 'quux', 'corge']

>>> a.pop(-3)
'qux'
>>> a
['foo', 'baz', 'quux', 'corge']
```

Remove an Object from List (Cont.)

- If index is omitted, **pop( )** simply removes the last item in the list.

```python
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> a.pop()
'corge'
>>> a
['foo', 'bar', 'baz', 'qux', 'quux']

>>> a.pop()
'quux'
>>> a
['foo', 'bar', 'baz', 'qux']
```

Remove an Object from List (Cont.)

- A list item can be deleted with the **del** command

Python

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> del a[3]
>>> a
['foo', 'bar', 'baz', 'quux', 'corge']
```

List Slicing

- If a is a list, the expression a[m:n] returns the portion of a from index m to, but not including, index n:

Python

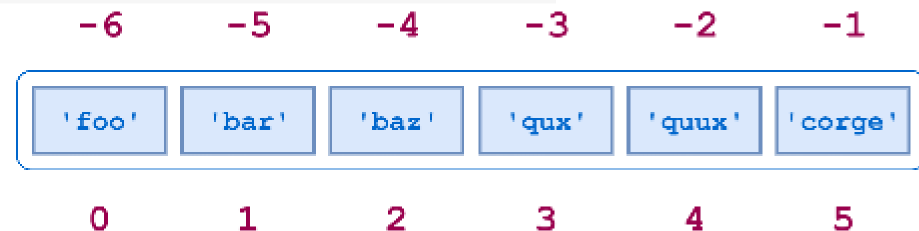
```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> a[2:5]
['baz', 'qux', 'quux']
```

- Both positive and negative indices can be specified:

Python

```
>>> a[-5:-2]
['bar', 'baz', 'qux']
>>> a[1:4]
['bar', 'baz', 'qux']
>>> a[-5:-2] == a[1:4]
True
```



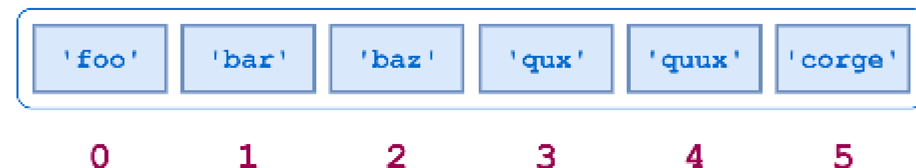
List Slicing (Cont.)

- Omitting the first index starts the slice at the beginning of the list, and omitting the second index extends the slice to the end of the list:

Python

```python
>>> print(a[:4], a[0:4])
['foo', 'bar', 'baz', 'qux'] ['foo', 'bar', 'baz', 'qux']
>>> print(a[2:], a[2:len(a)])
['baz', 'qux', 'quux', 'corge'] ['baz', 'qux', 'quux', 'corge']

>>> a[:4] + a[4:]
['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> a[:4] + a[4:] == a
True
```

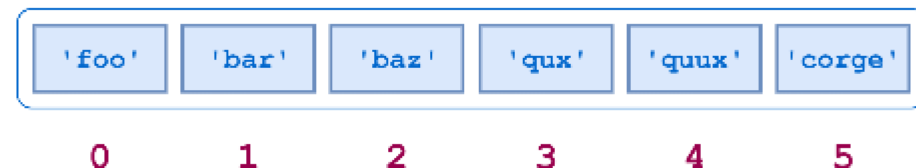


List Slicing (Cont.)

- A third index designates a stride (step) can be added to slicing, which indicates how many characters to jump after retrieving each character in the slice.
- You can specify a stride either positive or negative:

Python

```python
>>> a[0:6:2]
['foo', 'baz', 'quux']
>>> a[1:6:2]
['bar', 'qux', 'corge']
>>> a[6:0:-2]
['corge', 'qux', 'bar']
```



List Slicing (Cont.)

- The [:] syntax works for lists.
  - If s is a string, s[:] returns a reference to the same object:

Python

```python
>>> s = 'foobar'
>>> s[:]
'foobar'
>>> s[:] is s
True
```

- Conversely, if a is a list, a[:] returns a new object that is a copy of a:

Python

```python
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> a[:]
['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> a[:] is a
False
```

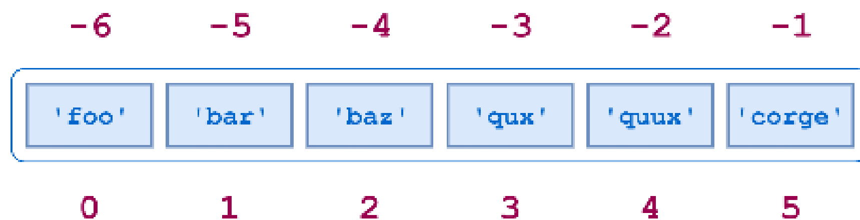
Modifying a Single List Value

- A single value in a list can be replaced by indexing and simple assignment

Python

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> a
['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> a[2] = 10
>>> a[-1] = 20
>>> a
['foo', 'bar', 10, 'qux', 'quux', 20]
```



Sub-List

- A list can be nested, and contain sub-list.

Python

```
>>> x = ['a', ['bb', ['ccc', 'ddd'], 'ee', 'ff'], 'g', ['hh', 'ii'], 'j']
```

- x[0], x[2], and x[4] are strings, each one character long

Python

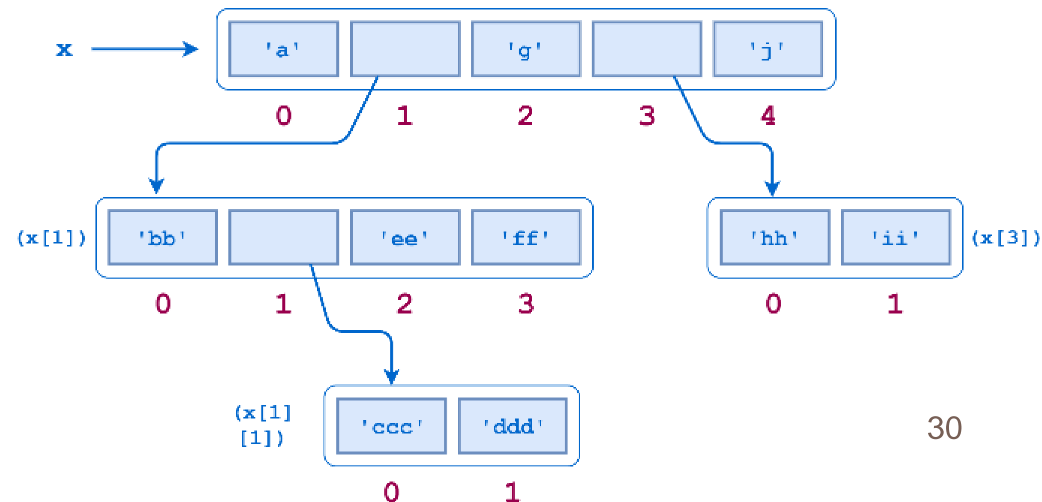
```
>>> print(x[0], x[2], x[4])
a g j
```

- x[1] and x[3] are sublists:

Python

```
>>> x[1]
['bb', ['ccc', 'ddd'], 'ee', 'ff']

>>> x[3]
['hh', 'ii']
```



Access Items in Sublist

- To access the items in a sublist, simply append an additional index:

Python

```
>>> x[1]
['bb', ['ccc', 'ddd'], 'ee', 'ff']

>>> x[1][0]
'bb'
>>> x[1][1]
['ccc', 'ddd']
>>> x[1][2]
'ee'
>>> x[1][3]
'ff'

>>> x[3]
['hh', 'ii']
>>> print(x[3][0], x[3][1])
hh ii
```

Python

```
>>> x[1][1]
['ccc', 'ddd']
>>> print(x[1][1][0], x[1][1][1])
ccc ddd
```

Python

```
>>> x[1][1][-1]
'ddd'
>>> x[1][1:3]
[['ccc', 'ddd'], 'ee']
>>> x[3][::-1]
['ii', 'hh']
```

Function Scope in List

- Operators and functions apply to only the list at the level you specify and are not recursive.

Python

```python
>>> x
['a', ['bb', ['ccc', 'ddd'], 'ee', 'ff'], 'g', ['hh', 'ii'], 'j']
>>> len(x)
5
```

x has only five elements: three strings and two sublists. The individual elements in the sublists don't count toward x's length.

Python

```python
>>> 'ddd' in x
False
>>> 'ddd' in x[1]
False
>>> 'ddd' in x[1][1]
True
```

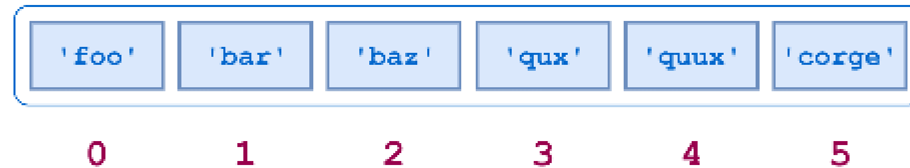
'ddd' is not one of the elements in x or x[1]. It is only directly an element in the sublist x[1][1]. An individual element in a sublist does not count as an element of the parent list(s).

Modifying Multiple List Values

- Python allows change several contiguous elements in a list at one time with slice assignment, which has the following syntax:

Python

```
a[m:n] = <iterable>
```



Python

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> a[1:4]
['bar', 'baz', 'qux']
>>> a[1:4] = [1.1, 2.2, 3.3, 4.4, 5.5]
>>> a
['foo', 1.1, 2.2, 3.3, 4.4, 5.5, 'quux', 'corge']
>>> a[1:6]
[1.1, 2.2, 3.3, 4.4, 5.5]
>>> a[1:6] = ['Bark!']
>>> a
['foo', 'Bark!', 'quux', 'corge']
```

Python

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> a[1:5] = []
>>> a
['foo', 'corge']

>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> del a[1:5]
>>> a
['foo', 'corge']
```

Tuple

Tuple

- A tuple is a sequence of values, which can be of any type and they are indexed by integer.
- Tuples are just like list, but we can't change values of tuples in place.
  - Tuples are immutable.
- The index value of tuple starts from 0.
- A tuple consists of a number of values separated by commas.
  - For example (10, 20, 30, 40)

Tuple

- Tuple is an order collection of objects identical to lists in all respects, except for the following properties:
  - Tuples are defined by enclosing the elements in parentheses ( ).
  - Tuples are immutable.

Python

```
>>> t = ('foo', 'bar', 'baz', 'qux', 'quux', 'corge')
>>> t
('foo', 'bar', 'baz', 'qux', 'quux', 'corge')

>>> t[0]
'foo'
>>> t[-1]
'corge'
>>> t[1::2]
('bar', 'qux', 'corge')
```

Why use Tuple instead of List?

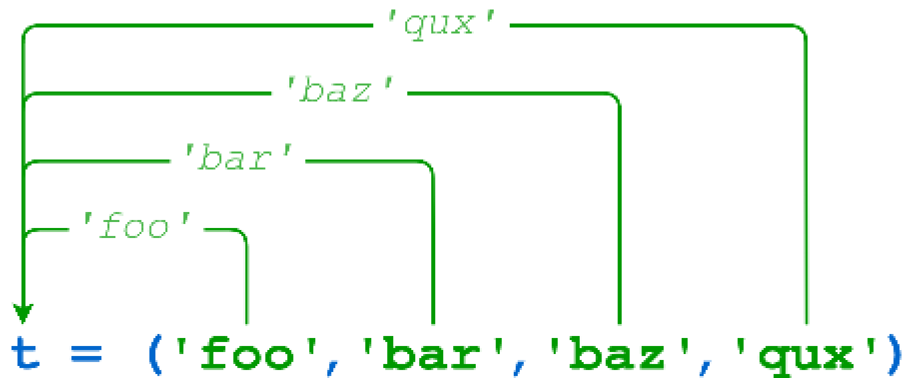
- Program execution is faster when manipulating a tuple than it is for the equivalent list.
- You don’t want data to be modified.
  - If the values in the collection are meant to remain constant for the life of the program, using a tuple instead of a list guards against accidental modification.
- Dictionary requires its components as immutable type.
  - A tuple can be used for this purpose, whereas a list can’t be.

Tuple Packing

- A literal tuple containing several items can be assigned to a single object:

Python

```
>>> t = ('foo', 'bar', 'baz', 'qux')
```

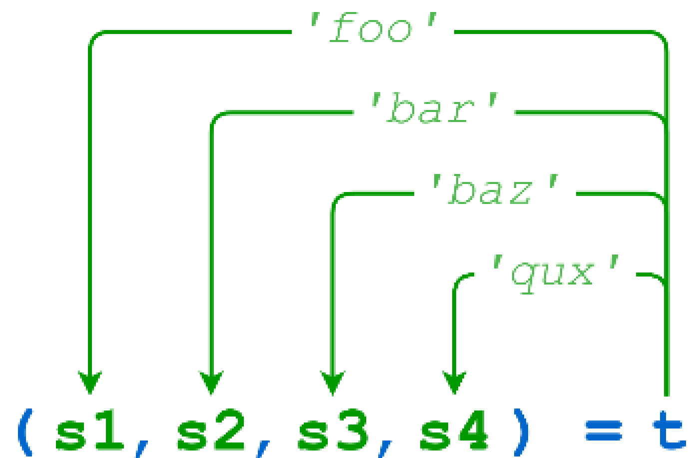


Tuple Unpacking

- If that “packed” object is subsequently assigned to a new tuple, the individual items are “unpacked” into the objects in the tuple:

Python

```
>>> (s1, s2, s3, s4) = t
>>> s1
'foo'
>>> s2
'bar'
>>> s3
'baz'
>>> s4
'qux'
```



Example

- Packing and unpacking can be combined into one statement to make a compound assignment:

Python

```
>>> (s1, s2, s3, s4) = ('foo', 'bar', 'baz', 'qux')
>>> s1
'foo'
>>> s2
'bar'
>>> s3
'baz'
>>> s4
'qux'
```

Example (Cont.)

- In assignments like this and a small handful of other situations, Python allows the parentheses that are usually used for denoting a tuple to be left out:

Python

```
>>> t = 1, 2, 3
>>> t
(1, 2, 3)

>>> x1, x2, x3 = t
>>> x1, x2, x3
(1, 2, 3)

>>> x1, x2, x3 = 4, 5, 6
>>> x1, x2, x3
(4, 5, 6)

>>> t = 2,
>>> t
(2,)
```

Swap in Tuple

- In Python, the swap can be done with a single tuple assignment

Python

```
>>> a = 'foo'
>>> b = 'bar'
>>> a, b
('foo', 'bar')

>>># Magic time!
>>> a, b = b, a

>>> a, b
('bar', 'foo')
```

Set

Set

- A set is an unordered collection of items. Every set element is unique (no duplicates) and must be immutable (cannot be changed).
  - Sets are unordered.
  - Set elements are unique. Duplicate elements are not allowed.
  - A set itself may be modified, but the elements contained in the set must be of an immutable type.

Creating Sets

- A set is created by placing all the items (elements) inside curly braces {}, separated by comma, or by using the built-in ***set( )*** function.
  - The argument to ***set( )*** is an iterable. It generates a list of elements to be placed into the set.
  - The objects in curly braces are placed into the set intact, even if they are iterable.

```
s = {1.0, "Hello", (1, 2, 3)}
s
```

```
{(1, 2, 3), 1.0, 'Hello'}
```

```
MyList = ["A", "C", "D", "B"]
MySet = set( MyList)
MySet
```

```
{'A', 'B', 'C', 'D'}
```

Set Size and Membership

- The ***len( )*** function returns the number of elements in a set, and the in and not in operators can be used to test for membership:

```
MySet = {"A", "B", "C"}
len(MySet)
```

```
3
```

```
"A" in MySet
```

```
True
```

```
"Z" in MySet
```

```
False
```

Union

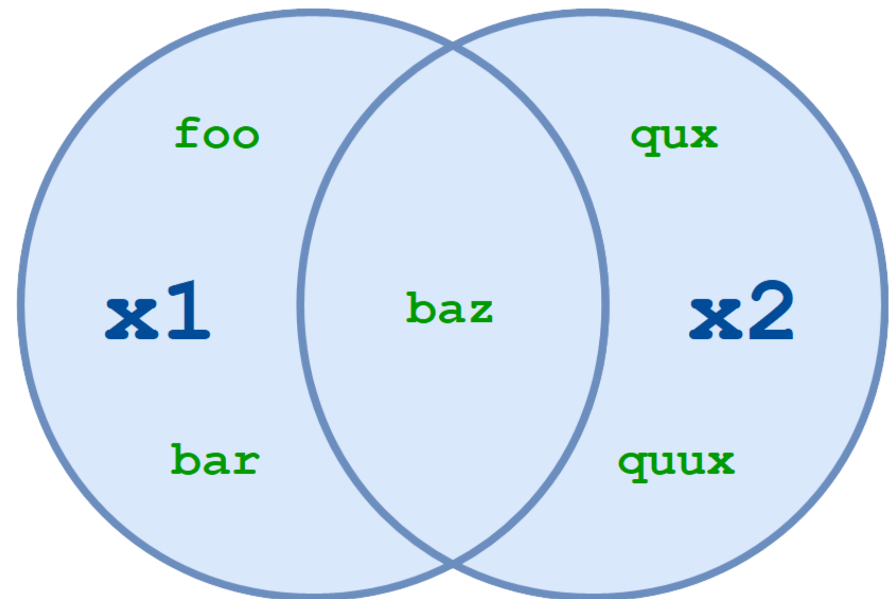
- x1 | x2 both return the set of all elements in either x1 or x2:
- Example:

```
[ ]  x1 = {"foo", "bar", "baz"}
     x2 = {"baz", "qux", "quux"}
     x1.union(x2)
```

```
{'bar', 'baz', 'foo', 'quux', 'qux'}
```

```
[ ]  x1 = {"foo", "bar", "baz"}
     x2 = {"baz", "qux", "quux"}
     x1 | x2
```

```
{'bar', 'baz', 'foo', 'quux', 'qux'}
```



Intersection

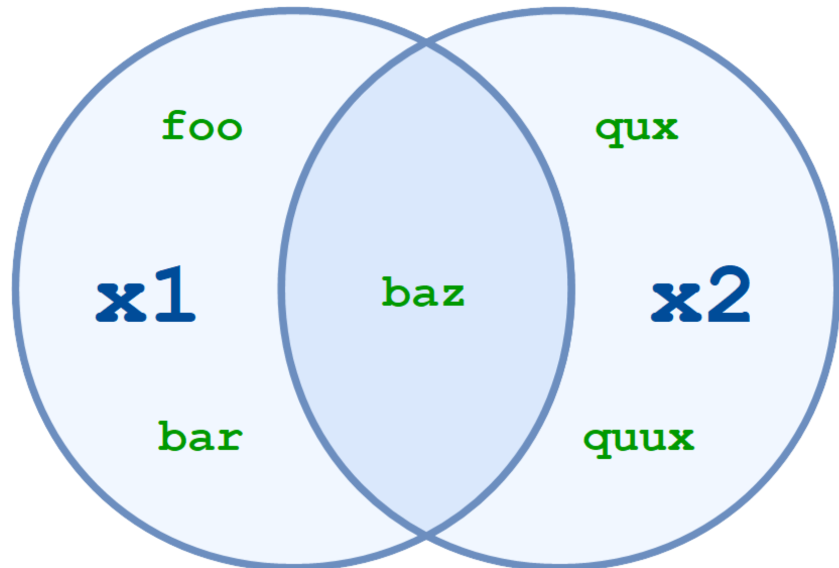
- x1 & x2 return the set of elements common to both x1 and x2:
- Example:

```
[1]  x1 = {"foo", "bar", "baz"}
     x2 = {"baz", "qux", "quux"}
     x1.intersection(x2)
```

```
{'baz'}
```

```
[2]  x1 = {"foo", "bar", "baz"}
     x2 = {"baz", "qux", "quux"}
     x1 & x2
```

```
{'baz'}
```



Difference

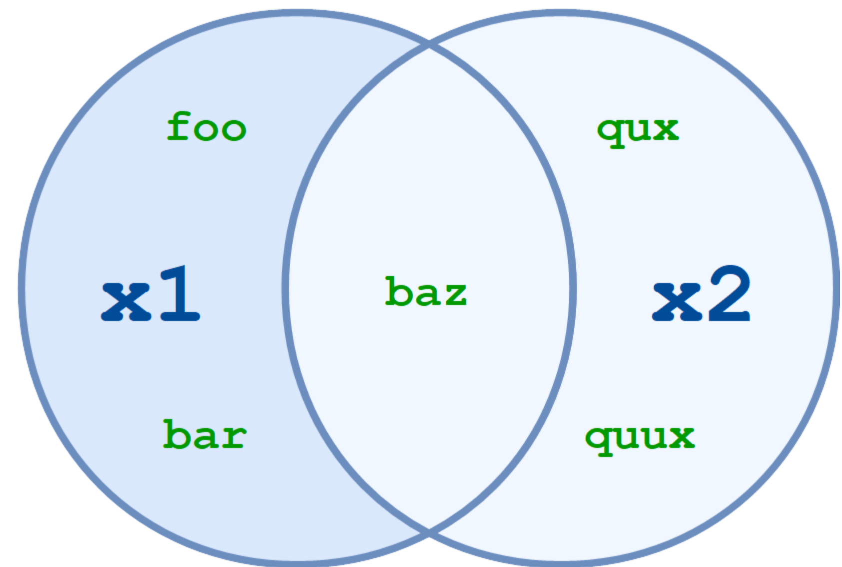
- x1 - x2 return the set of all elements that are in x1 but not in x2:
- Example:

```
x1 = {"foo", "bar", "baz"}
x2 = {"baz", "qux", "quux"}
x1.difference(x2)
```

```
{'bar', 'foo'}
```

```
x1 = {"foo", "bar", "baz"}
x2 = {"baz", "qux", "quux"}
x1 - x2
```

```
{'bar', 'foo'}
```



Symmetric Difference

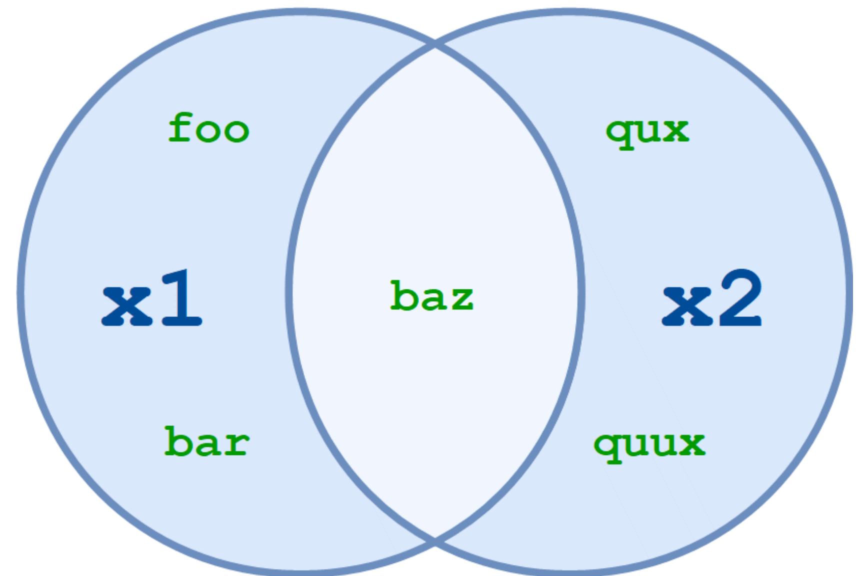
- x1 ^ x2 return the set of all elements in either x1 or x2, but not both:
- Example:

```
[ ] x1 = {"foo", "bar", "baz"}
    x2 = {"baz", "qux", "quux"}
    x1.symmetric_difference(x2)

{'bar', 'foo', 'quux', 'qux'}
```

```
[ ] x1 = {"foo", "bar", "baz"}
    x2 = {"baz", "qux", "quux"}
    x1 ^ x2

{'bar', 'foo', 'quux', 'qux'}
```



Dictionary

Dictionary

- Python provides another composite data type called a dictionary, which is similar to a list in that it is a collection of objects.
- Dictionaries and Lists share the following characteristics:
  - Both are mutable.
  - Both are dynamic. They can grow and shrink as needed.
  - Both can be nested. A dictionary can contain another dictionary. A dictionary can also contain a list, and vice versa.
- Dictionaries differ from lists in two important ways.
  - The first is the ordering of the elements:
    - Elements in a list have a distinct order, which is an intrinsic property of that list.
    - Dictionaries are unordered. Elements are not kept in any specific order.
  - The second difference lies in how elements are accessed:
    - List elements are accessed by their position in the list, via indexing.
    - Dictionary elements are accessed via keys.

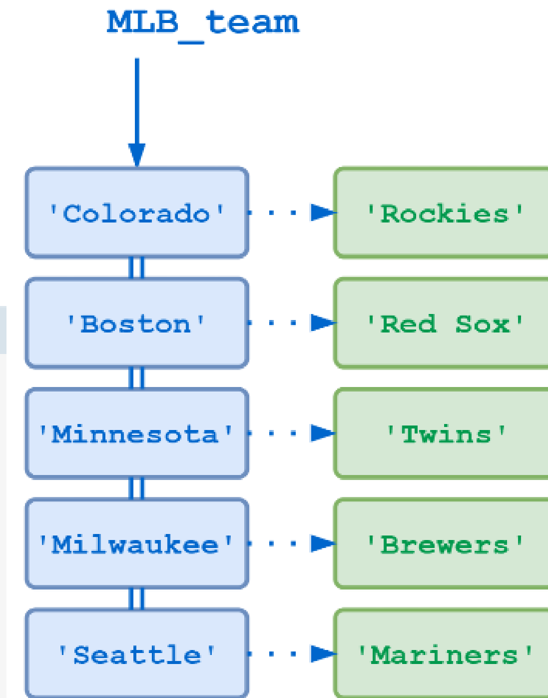
Defining a Dictionary

- Dictionaries are Python's implementation of a data structure that is more generally known as an associative array.
- A dictionary consists of a collection of key-value pairs.
  - Each key-value pair maps the key to its associated value.
- You can define a dictionary by enclosing a comma-separated list of key-value pairs in curly braces {}. A colon separates each key from its associated value

```python
d = {
    <key>: <value>,
    <key>: <value>,
      .
      .
      .
    <key>: <value>
}
```

Example

- The following example defines a dictionary that maps a location to the name of its corresponding Major League Baseball team



Python

```
>>> MLB_team = {
...     'Colorado' : 'Rockies',
...     'Boston'   : 'Red Sox',
...     'Minnesota': 'Twins',
...     'Milwaukee': 'Brewers',
...     'Seattle'  : 'Mariners'
... }
>>> type(MLB_team)
<class 'dict'>

>>> MLB_team
{'Colorado': 'Rockies', 'Boston': 'Red Sox', 'Milwaukee': 'Brewers',
'Seattle': 'Mariners', 'Minnesota': 'Twins'}
```

Define Dictionary using dict Statement

- Dictionary can be construct with the built-in ***dict( )*** function. The argument to ***dict( )*** should be a sequence of key-value pairs

Python

```
d = dict([
    (<key>, <value>),
    (<key>, <value),
      .
      .
      .
    (<key>, <value>)
])
```

Example

- Dictionary can be defined using ***dict( )***

Python

```
>>> MLB_team = dict([
...     ('Colorado', 'Rockies'),
...     ('Boston', 'Red Sox'),
...     ('Minnesota', 'Twins'),
...     ('Milwaukee', 'Brewers'),
...     ('Seattle', 'Mariners')
... ])
```

- If the key values are simple strings, they can be specified as keyword arguments.

Python

```
>>> MLB_team = dict(
...     Colorado='Rockies',
...     Boston='Red Sox',
...     Minnesota='Twins',
...     Milwaukee='Brewers',
...     Seattle='Mariners'
... )
```

Accessing Dictionary Values

- A value is retrieved from a dictionary by specifying its corresponding key in square brackets [ ]:

Python

```
>>> MLB_team['Minnesota']
'Twins'
>>> MLB_team['Colorado']
'Rockies'
```

Adding Key and Value to Dictionary

- Adding an entry to an existing dictionary is simply a matter of assigning a new key and value

Python

```
>>> MLB_team['Kansas City'] = 'Royals'
>>> MLB_team
{'Colorado': 'Rockies', 'Boston': 'Red Sox', 'Milwaukee': 'Brewers',
'Seattle': 'Mariners', 'Minnesota': 'Twins', 'Kansas City': 'Royals'}
```

Updating Value in Dictionary

- If you want to update an entry, you can just assign a new value to an existing key:

Python

```
>>> MLB_team['Seattle'] = 'Seahawks'
>>> MLB_team
{'Colorado': 'Rockies', 'Boston': 'Red Sox', 'Milwaukee': 'Brewers',
'Seattle': 'Seahawks', 'Minnesota': 'Twins', 'Kansas City': 'Royals'}
```

Delete Value from Dictionary

- To delete an entry, use the ***del*** statement, specifying the key to delete:

Python

```
>>> del MLB_team['Seattle']
>>> MLB_team
{'Colorado': 'Rockies', 'Boston': 'Red Sox', 'Milwaukee': 'Brewers',
'Minnesota': 'Twins', 'Kansas City': 'Royals'}
```

Restrictions on Dictionary Keys

- There are a couple restrictions that dictionary keys must abide by.
  - A given key can appear in a dictionary only once. Duplicate keys are not allowed.
- A dictionary key must be of a type that is immutable.
  - That means an integer, float, string, or Boolean can be a dictionary key.
  - A tuple can also be a dictionary key, because tuples are immutable:

Operators and Built-in Functions

- The *in* and *not* in operators return True or False according to whether the specified operand occurs as a key in the dictionary

Python

```
>>> MLB_team = {
...     'Colorado' : 'Rockies',
...     'Boston'   : 'Red Sox',
...     'Minnesota': 'Twins',
...     'Milwaukee': 'Brewers',
...     'Seattle'  : 'Mariners'
... }

>>> 'Milwaukee' in MLB_team
True
>>> 'Toronto' in MLB_team
False
>>> 'Toronto' not in MLB_team
True
```

Operators and Built-in Functions (Cont.)

- The ***len( )*** function returns the number of key-value pairs in a dictionary:

Python

```
>>> MLB_team = {
...     'Colorado' : 'Rockies',
...     'Boston'   : 'Red Sox',
...     'Minnesota': 'Twins',
...     'Milwaukee': 'Brewers',
...     'Seattle'  : 'Mariners'
... }
>>> len(MLB_team)
5
```

Built-in Dictionary Methods

Method	Description
clear()	Clears a dictionary
get(<*key*>[, <*default*>])	Returns the value for a key if it exists in the dictionary
items()	Returns a list of key-value pairs in a dictionary
keys()	Returns a list of keys in a dictionary
values()	Returns a list of values in a dictionary
pop(<*key*>[, <*default*>])	Removes a key from a dictionary, if it is present, and returns its value
popitem()	Removes a key-value pair from a dictionary
update(<*obj*>)	Merges a dictionary with another dictionary or with an iterable of key-value pair

Clears a Dictionary

- ***clear( )*** empties dictionary of all key-value pairs

Python

```
>>> d = {'a': 10, 'b': 20, 'c': 30}
>>> d
{'a': 10, 'b': 20, 'c': 30}

>>> d.clear()
>>> d
{}
```

Get the Dictionary value by Key

- The ***get( )*** method provides a convenient way of getting the value of a key from a dictionary without checking ahead of time whether the key exists, and without raising an error.

```
Python
>>> d = {'a': 10, 'b': 20, 'c': 30}

>>> print(d.get('b'))
20
>>> print(d.get('z'))
None
```

get(<key>) searches dictionary for <key> and returns the associated value if it is found.
If <key> is not found, it returns None:

```
Python
>>> print(d.get('z', -1))
-1
```

If <key> is not found and the optional <default> argument is specified, that value is returned instead of None

Get the List of Tuples in Dictionary

- ***items( )*** returns a list of tuples containing the key-value pairs in dictionary. The first item in each tuple is the key, and the second item is the key's value

Python

```
>>> d = {'a': 10, 'b': 20, 'c': 30}
>>> d
{'a': 10, 'b': 20, 'c': 30}

>>> list(d.items())
[('a', 10), ('b', 20), ('c', 30)]
>>> list(d.items())[1][0]
'b'
>>> list(d.items())[1][1]
20
```

Get the List of Key in Dictionary

- ***keys( )*** returns a list of all keys in dictionary

Python

```
>>> d = {'a': 10, 'b': 20, 'c': 30}
>>> d
{'a': 10, 'b': 20, 'c': 30}

>>> list(d.keys())
['a', 'b', 'c']
```

Get the List of Values in Dictionary

- ***values( )*** returns a list of all values in dictionary

Python

```
>>> d = {'a': 10, 'b': 20, 'c': 30}
>>> d
{'a': 10, 'b': 20, 'c': 30}

>>> list(d.values())
[10, 20, 30]
```

Python

```
>>> d = {'a': 10, 'b': 10, 'c': 10}
>>> d
{'a': 10, 'b': 10, 'c': 10}

>>> list(d.values())
[10, 10, 10]
```

Any duplicate values in dictionary will be returned as many times as they occur

Remove from Dictionary by Key

- If *<key>* is present in dictionary, ***pop(****<key>****)*** removes *<key>* and returns its associated value

Python

```
>>> d = {'a': 10, 'b': 20, 'c': 30}

>>> d.pop('b')
20
>>> d
{'a': 10, 'c': 30}
```

Python

```
>>> d = {'a': 10, 'b': 20, 'c': 30}
>>> d.pop('z', -1)
-1
>>> d
{'a': 10, 'b': 20, 'c': 30}
```

If <key> is not in d, and the optional <default> argument is specified, then that value is returned, and no exception is raised

Remove Pair from Dictionary

- The ***popitem( )*** returns and removes an arbitrary element (key, value) pair from the dictionary.

Python

```
>>> d = {'a': 10, 'b': 20, 'c': 30}

>>> d.popitem()
('c', 30)
>>> d
{'a': 10, 'b': 20}

>>> d.popitem()
('b', 20)
>>> d
{'a': 10}
```

Merge Dictionary

- If <*obj*> is a dictionary, update(<*obj*>) merges the entries from <*obj*> into dictionary.
- For each key in <*obj*>:
  - If the key is not present in dictionary, the key-value pair from <*obj*> is added to dictionary.
  - If the key is already present in dictionary, the corresponding value in dictionary for that key is updated to the value from <*obj*>.

Python

```
>>> d1 = {'a': 10, 'b': 20, 'c': 30}
>>> d2 = {'b': 200, 'd': 400}

>>> d1.update(d2)
>>> d1
{'a': 10, 'b': 200, 'c': 30, 'd': 400}
```

Key 'b' already exists in d1, so its value is updated to 200, the value for that key from d2. However, there is no key 'd' in d1, so that key-value pair is added from d2

Merge Dictionary (Cont.)

- *<obj>* may also be a sequence of key-value pairs, similar to when the ***dict( )*** function is used to define a dictionary.

Python

```
>>> d1 = {'a': 10, 'b': 20, 'c': 30}
>>> d1.update([('b', 200), ('d', 400)])
>>> d1
{'a': 10, 'b': 200, 'c': 30, 'd': 400}
```

<obj> can be specified as a list of tuples

- The values to merge can be specified as a list of keyword arguments

Python

```
>>> d1 = {'a': 10, 'b': 20, 'c': 30}
>>> d1.update(b=200, d=400)
>>> d1
{'a': 10, 'b': 200, 'c': 30, 'd': 400}
```

Summary

Summary

Data Structure	Ordered	Mutable	Constructor	Example
List	Yes	Yes	`[ ]` or `list()`	`[5.7, 4, 'yes', 5.7]`
Tuple	Yes	No	`( )` or `tuple()`	`(5.7, 4, 'yes', 5.7)`
Set	No	Yes	`{}`* or `set()`	`{5.7, 4, 'yes'}`
Dictionary	No	Yes**	`{ }` or `dict()`	`{'Jun': 75, 'Jul': 89}`