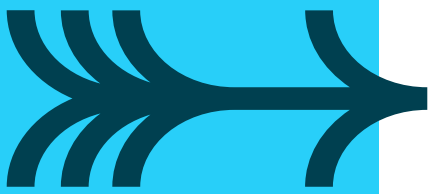
POWERING POTENTIAL

Python 3 Programming

Collections

COLLECTIONS



Contents

This chapter discusses more basic building blocks of a Python program - its container variable types - collectively known as collections.

Python types - reminder

Built in sequence types:

- Strings (`str`)

```
'Norwegian Blue', "Mr. Khan's bike"
```

- Lists (`list`)

```
['Cheddar', ['Camembert', 'Brie'], 'Stilton']
```

- Tuples (`tuple`)

```
(47, 'Spam', 'Major', 683, 'Ovine Aviation')
```

- We also have bytearray (read/write) and bytes (read only)
- **Used for binary data**

Not all collections are sequences

- A set is an *unordered* collection of *unique* objects
- Dictionaries are a special form of set

```
{'Totnes':'Barber', 'BritishColumbia':'Lumberjack'}
```

Strings, Lists and Tuples are ordered collections of objects, also known as sequences.
The types bytearray and bytes are used for raw binary data, and were introduced in Python 2.6. They are similar to strings, and can be accessed using the same methods as strings and lists, and may be indexed using []. They hold 8-bit signed integers in the range 0-255.
Dictionaries are collections of objects accessed by key, and are similar to associative arrays in awk and PHP, and hashes in Perl and Ruby.
Sets were introduced into Python 2.4.
There is also a collections module in the Python Standard Library, this includes alternative base classes for the default containers, as well as some other types, such as deque, and in Python 3.1 the OrderedDict type was added.

Generic built-in functions

- **Most *iterables* support the `len`, `min`, `max`, and `sum` built-in functions**
  - `Len` Number of elements
  - `min` Minimum value
  - `max` Maximum value
  - `sum` Numeric summation (not string or byte objects)
- **String and byte objects do not support sum**
- **Dictionaries implement `min`, `max`, and `sum` on keys**
- **The `sum` built-in will raise a TypeError if the item is not a number**

```
myn = [45, 66, 12, 3, 99, 3.142, 42]
print("min:", min(myn), "max:", max(myn))
print("sum:", sum(myn))

myd = {'fred':3, 'jim':8, 'dave':42}
print("min:", min(myd), "max:", max(myd))
```

```
min: 4 max: 99
sum: 270.142
min: dave max jim
```

The example code produces the following output

```
min: 3 max: 99
sum: 270.142
min: dave max: jim
```

Range objects can also be included in the list of sequence objects.
sum supports an optional second parameter, start, which gives the initial value of the sum (defaults to zero).
The **len** function returns the number of characters when used with a string, and the number of bytes when used with a bytes object.
For example:

```
mys = chr(0x20ac)
print(mys, len(mys))
myb = mys.encode()
print(myb, len(myb))
```

Gives:

```
€ 1
b'\xe2\x82\xac' 3
```

Useful tuple operations

Swap references

```
a, b = b, a
```

Set values from a numeric range

```
Gouda, Edam, Caithness = range(3)
```

```
0 1 2
```

py3

Repeat values

```
mytuple = 'a', 'b', 'c'
another = mytuple * 4
```

```
('a', 'b', 'c', 'a', 'b', 'c', 'a', 'b', 'c', 'a', 'b', 'c')
```

Be careful of single values and the trailing comma

```
thing = ('Hello')
print(type(thing))
thing = ('Hello',)
print(type(thing))
```

```
<class 'str'>
<class 'tuple'>
```

Tuples *elements* can be assigned as long as they are named variables. The example shown, swaps two variables (`a` and `b`) but many more variables could be involved. This is just a special case of assigning one tuple to another. Notice we are not altering a tuple, we are altering the values of the variables within. There must be the same number of variables on the left as there are values on the right. The parentheses are optional, so the example could also be written as:

```
(a, b) = (b, a)
```

The second example of a tuple operation sets three variables and gives them values from a range. `Gouda` will have the value 0, `Edam` will be 1, and `Caithness` 2.

In Python 2 the **`range()`** function created a complete temporary list of integers in memory, so the `xrange()` function was used instead. In Python 3 **`range()`** returns a *lazy list* (as `xrange()` did), which only generates the next value when it is needed.

Tuples (and strings and lists) may be repeated using the **`*`** operator. In this case, the tuple `another` will have the values of `mytuple` repeated 4 times.

It is tempting to use parentheses around a single item, but this will

not produce a tuple. A trailing comma is required.

Python lists

- **Python lists are similar to arrays in other languages***
  - Can contain objects of any type
  - Multi-dimensional lists are just lists containing references to other lists
- **Create a list using `list(`*object*`)` or `[]`**
- **Access list elements using `[]` or by method calls**
  - Indexes on the left start at zero
  - Indexes on the right start at -1

```
cheese = ['Cheddar', 'Stilton', 'Cornish Yarg']
print(cheese[1])
cheese[-1] = 'Red Leicester'
print(cheese)
```

```
Stilton
['Cheddar', 'Stilton', 'Red Leicester']
```

- Multiply operator * can also be applied to a list

Lists are objects containing a sequential collection of other objects, commonly called *elements*. Elements may be accessed by a position (counting from zero) specified within **`[ ]`** which is probably familiar from other languages.

Lists are *Mutable*, that is they may be changed, so they are similar to arrays in some languages. They are dynamic in that they may be extended or shrunk. New items may be added anywhere with the list, and also removed (discussed in later slides).

List may be concatenated (joined together) using the + operator.

***Python lists are similar to arrays in other languages**

Don't carry this analogy too far. Python lists are not like arrays in C. That's a good thing in general programming, but carries a performance overhead – C arrays are very fast. So Python has a module called **`array`** in the standard library which provides objects that are very similar to C arrays.

Tuple and list slicing

- **Slice by start and end *position***
- Counting from zero on lhs, from -1 on rhs

```
mytuple=('eggs', 'bacon', 'spam', 'tea', 'beans')
print(mytuple[2:4])
('spam', 'tea')
print(mytuple[-4])
bacon
mylist = list(mytuple)
print(mylist[1:])
['bacon', 'spam', 'tea', 'beans']
print(mylist[:2])
['eggs', 'bacon']
```

start

end+1

- List elements may be removed using `del`

```
cheese = ['Cheddar', 'Camembert', 'Brie', 'Stilton']
del cheese[1:3]
print(cheese)
```

```
['Cheddar', 'Stilton']
```

Sequential composite objects like strings, tuples, and lists may be sliced. Often only one item is required, in which case the syntax uses the familiar square brackets with the index inside. Items are indexed from zero on the left, or -1 (with a negative count) from the right.

To slice a range of elements we specify the start index, a colon, then end index plus one. That is, the slice is taken up to, but not including, the second index position.

If the start index is not given, then the default is zero (first element). For example, string[:-1] will give the characters up to, but not including, the last character in a string, effectively deleting it. Defaulting the second index slices to the end of the object.

Python strings may be sliced in a similar way.

The del statement can delete a slice, a comma separated list of elements, or the whole list. Notice that del is a statement, not a built-in function, so parentheses are not required around the name being deleted.

Extended iterable unpacking

- **Python 3 allows unpacking to a wildcard**
  - Only allowed on the left-side of an assignment

py3

```
mytuple = 'eggs', 'bacon', 'spam', 'tea'
x, y, z = mytuple
```

```
ValueError: too many values to unpack
```

```
mytuple = 'eggs', 'bacon', 'spam', 'tea'
x, y, *z = mytuple
print(x, y, z)
```

```
eggs bacon ['spam', 'tea']
```

```
t1 = 'cat', 'dog', 'python', 'mouse', 'camel'
t2 = 'kelp', 'crab', 'lobster', 'fish'
for a, *b, c in t1, t2:
    print(a, b, c)
```

```
cat ['dog', 'python', 'mouse'] camel
kelp ['crab', 'lobster'] fish
```

PEP 3132 introduced a welcome simplification of the syntax when assigning to tuples - unpacking. In the past, if assigning to variables in a tuple, the number of items on the left of the assignment must be exactly equal to that on the right.
In Python 3, we can designate any variable on the left as a tuple by prefixing with an asterisk *. That will grab as many values as it can, as a list, while still populating the variables to its right (so it need not be the rightmost item). This avoids many nasty slices when we don't know the length of a tuple.

Another form of unpacking, also using a *, has been available for some time, including Python 2, in function arguments, which we shall see later.

Adding items to a list

On the left

```
cheese[:0] = ['Cheshire', 'Ilchester']
```

On the right

```
cheese += ['Oke', 'Devon Blue']
cheese.extend(['Oke', 'Devon Blue'])
```

Same effect

- append can only be used for one item

```
cheese.append('Oke')
```

Anywhere

```
cheese = ['Cheddar', 'Stilton', 'Cornish Yarg']
cheese.insert(2, 'Cornish Brie')
cheese[2:2] = ['Cornish Brie']
print(cheese)
```

Same effect

```
['Cheddar', 'Stilton', 'Cornish Brie', 'Cornish Yarg']
```

List may be extended in any direction from any position. In the first example, 'Cheshire' and 'Ilchester' and added to the front of the `cheese` list. In the second, we show two ways of adding items to the end of a list, using the **`+=`** operator and using the **`extend`** method. In theory, **`extend`** is more efficient, but it is unlikely that you would notice, or even be able to measure, a difference.
The third example shows the **`append`** method, which can only to used to add one item - but that is often enough.
Finally, we show two ways of inserting an item at a specific position - using the **`insert`** method (specifying the index position) and using a slice. Note that with **`insert`** we can only insert one item, but using a slice we can insert as many items as we wish.

Removing items by position

Use `pop(`*index*`)`

- The index number is optional, default -1 (rightmost item)
- Returns the deleted item

```
cheese = ['Cheddar', 'Stilton', 'Cornish Yarg']
saved = cheese.pop(1)
print("Saved1:", saved,", Result:", cheese)

saved = cheese.pop()
print("Saved2:", saved,", Result:", cheese)
```

```
Saved1: Stilton , Result: ['Cheddar', 'Cornish Yarg']
Saved2: Cornish Yarg , Result: ['Cheddar']
```

Remember that `del` may also be used

- Does not return the deleted item
- May delete more than one item by using a slice

The **`pop`** method removes the specified item from a list, the default being the last (rightmost) item. For example, use `cheese.pop(0)` to remove the leftmost item from the list.
An advantage of **`pop`** over **`del`** is that it returns the deleted item, on the other-hand **`del`** may remove more than one. Deleting from the right-hand end of the list, which **`pop`** does by default, is generally more efficient. Deleting from anywhere else, can mean that the internal representation of the list has to be rebuilt.

Removing list items by content

Use the `remove` method

- Removes the leftmost item matching the value

```
cheese = ['Cheddar', 'Stilton', 'Cornish Yarg',
          'Oke', 'Devon Blue']
cheese.remove('Oke')
print(cheese)
```

```
['Cheddar', 'Stilton', 'Cornish Yarg', 'Devon Blue']
```

Raises an exception if the item is not found

- Exceptions will be handled later...

```
cheese.remove('Brie')
```

```
Traceback (most recent call last):
  File "...", line 57, in <module>
    cheese.remove ('Brie')
ValueError: list.remove(x): x not in list
```

Sometimes we might not know the position of an item, we might want a "search and destroy" method - which is exactly what **`remove`** does. Note that data items in a list need not be unique, and **`remove`** will find the leftmost occurrence of the value.

Sorting

`sorted` built-in and `sort` method

- `sorted` can sort any *iterable* (often a *sequence*)
- `sorted` returns a sorted list - regardless of the original type
- `sort` sorts a list in-place
- Both have the following optional named parameters

key=*sort_key*	Function which takes a single argument
reverse=True	Default is False

```
cheese = ['Cornish Yarg', 'Oke', 'Edam', 'Stilton']
cheese.sort(key=len)
print(cheese)
```

```
['Oke', 'Edam', 'Stilton', 'Cornish Yarg']
```

```
nums = ['1001', '34', '3', '77', '42', '9', '87']
newstr = sorted(nums)
newnum = sorted(nums, key=int)
```

```
newstr: ['1001', '3', '34', '42', '77', '87', '9']
newnum: ['3', '9', '34', '42', '77', '87', '1001']
```

The sort algorithm used by Python is the Adaptive Stable Mergesort (algorithm by Tim Peters).
`sorted` was introduced in Python 2.4 and returns a sorted list. **`sort`** alters the list in-place, it returns `None`.
The key is a single argument function to be called which returns the key value. This is often a **`lambda`** function- an anonymous inline function, discussed later.
The **key** function is called once for each element to be sorted, and determines the actual value to be compared. Note that in the second set of examples, with the numbers, the first result is from a textual comparison and the second is numeric. If the values had not been enclosed with quotes then the default comparison would have worked correctly.
Old versions of Python **`sort`** and **`sorted`** also had a `cmp` argument (and there was a `cmp` built-in).
The `sorted()` built-in can sort anything which is iterable, which includes sequence types like tuples and strings, but always returns a list (remember that tuples and strings are immutable).
A `sort()` method should be described for all *mutable* sequence types. In the base types (those not requiring an external module)

that only includes **lists** and **bytearrays**.

Miscellaneous list methods

Count	*list*`.count('`*value*`')`	Return the number of occurrences of '*value*'
Index	*list*`.index('`*value*`')`	Return index position of leftmost '*value*'
Reverse	*list*`.reverse()`	Reverse a list in place

```
cheese = ['Cheddar', 'Cheshire', 'Stilton', 'Cheshire']
print(cheese.count('Cheshire'))
print(cheese.index('Cheshire'))
cheese.reverse()
print(cheese)
```

```
2
1
['Cheshire', 'Stilton', 'Cheshire', 'Cheddar']
```

A full table of list methods is shown on the next slide.
The `index` method returns the index position of the leftmost item found (counting from zero). Throws a ValueError exception if the item is not found.

List methods

list.append(*item*)	Append *item* to the end of *list*
list.clear()	Remove all items from *list* (3.3)
list.count(*item*)	Return number of occurrences of *item*
list.extend(*items*)	Append *items* to the end of *list* (as +=)
list.index(*item, start, end*)	Return the position of *item* in the *list*
list.insert(*position*, *item*)	Insert *item* at *position* in *list*
list.pop()	Remove and return last item in *list*
list.pop(*position*)	Remove and return item at *position* in *list*
list.remove(*item*)	Remove the first *item* from the *list*
list.reverse()	Reverse the *list* in-place
list.sort(…)	Sort the *list* in-place - arguments are the same as `sorted()`

py3

This slide is for reference. The clear() method was added at Python 3.3 and is not in Python 2.

Sets

A set is an *unordered* container of object references

- A set is *mutable*, a frozenset is *immutable*
- Set items are unique

Creating a set

- Any iterable type may be used

py3

```
s1 = {5, 6, 7, 8, 5}
print(s1)

s2 = set([9, 10, 11, 12, 9])
print(s2)

s3 = frozenset([9, 10, 11, 12, 9])
print(s3)
```

Python versions >= 2.7
Not <= 2.6

Python versions >= 2.4

```
{8, 5, 6, 7}
{9, 10, 11, 12}
frozenset({9, 10, 11, 12})
```

The format when printing
a set changed at Python 3

Sets can be considered to be like lists only being unordered. Somewhat like one half (the keys) of a dictionary. Indeed, we shall see later that the *dict*.**`keys()`** method can be treated as a set. They are useful for lookup tables, where an associated value is not needed and only membership need be tested (using **in**), and for operations between sets, like intersection.
Notice that any original order is lost.
The method **`add`** is used to add a single element. To add multiple elements, use **`update`**.
Note that the output shown is for Python 3, on Python 2.4 the set was shown inside `[]` with the prefix 'set'.

Set methods

Add using the `add` method, remove using `remove`

```
s4 = {23, 42, 66, 123}
s5 = {56, 27, 42}
print("{:20} {:20}".format(s4, s5))

s4.remove(123)
s5.add(123)
print("{:20} {:20}".format(s4, s5))
```

```
{66, 123, 42, 23}    {56, 42, 27}
{66, 42, 23}         {56, 123, 42, 27}
```

Other set methods:

- `Len` Return the number of elements in the set
- `discard` Remove element *if present*
- `pop` Remove and return the next element from the set
- `clear` Remove all elements

The method **`add`** is used to add a single element.
To add multiple elements, use **`update`**. There are three "updates", and alternative operators:

`update`	`|=`	Update the set from another
`intersection_update`	`&=`	Update the set, with common elements
`difference_update`	`-=`	Remove elements found in the other

Frozensets are immutable so do not have the `add`, `update`, or `remove` methods, but they are hashable so can be used as entries in other sets.

Exploiting sets

How do I remove duplicates from a list?

- But we lose the original order

```
cheese = ['Cheddar', 'Stilton', 'Cornish Yarg',
          'Oke', 'Stilton', 'Cheshire']
cheese = list(set(cheese))
```

list() is required, otherwise 'cheese' would now refer to a set

```
['Cornish Yarg', 'Cheshire', 'Cheddar', 'Stilton', 'Oke']
```

How do I remove several items from a list?

```
cheese = ['Cheddar', 'Stilton', 'Cornish Yarg',
          'Oke', 'Stilton', 'Cheshire']
cheese = list(set(cheese) - {'Stilton', 'Oke', 'Brie'})
```

```
['Cornish Yarg', 'Cheshire', 'Cheddar']
```

Sets can be used for a number of functions, including membership using **`in`**. Here, we have used them to exploit their features.
Items in a set are unique, so putting a list into a set will automatically remove any duplicates - the only problem being that any original order is lost. One reason is to remove duplicates before a sort. Without sets, we would use a dictionary for this, but with some 'throw-away' value.
We can apply various set operations, so taking a list and turning it into a set allow us to apply those operations to (what was) the list.
Notice that when we remove one set from another, the items in set on the right does not have to be in the left. In the example, 'Brie' does not exist in the set built from cheese, and it is not an error to try to remove it.

Set operators

- **Includes set operators and method calls**

Operator	Method	Returns a new set containing
`&`	`s6.intersection(s7)`	Each item that is in both sets
`|`	`s6.union(s7)`	All items in both sets
`-`	`s6.difference(s7)`	Items in s6 not in s7
`^`	`s6.symmetric_difference(s7)`	Items that occur in one set only

```
s6 = {23, 42, 66, 123}
s7 = {123, 56, 27, 42}

print(s6 & s7)
print(s6 | s7)
print(s6 - s7)
print(s6 ^ s7)
```

```
{42, 123}
{66, 27, 42, 23, 56, 123}
{66, 23}
{66, 23, 56, 27}
```

py3

The function versions can take any iterable as its parameter, so there is no need to convert to a set first.

Python dictionaries

Dictionaries are similar to sets but are accessed by keys

- Constructed from { }
  varname **=** **{***key1*:*object1*,*key2*:*object2*,*key3*:*object3*,…**}**
- Or using `dict()`
  varname **= dict(***key1*=*object1*,*key2*=*object2*,*key3*=*object3*,…**)**
- Accessed by key
  A key is usually a text string, or anything that yields a text string
  *varname***[***key***] =** *object*

```
mydict = {'Australia':'Canberra', 'Eire':'Dublin',
          'France':'Paris', 'Finland':'Helsinki',
          'UK':'London', 'US':'Washington'
}
print(mydict['UK'])

country = 'Iceland'
mydict[country] = 'Reykjavik'
```

Dictionaries are constructed from lists of key:object pairs, inside braces (curly brackets), although you may also assign them from the dict function, for example:

mydict = dict(Sweden = 'Stockholm', Norway = 'Oslo')

This form can only be used if the keys are text strings, not if they are numbers.

The value is an object of any valid class, including a list, tuple, or dictionary. No special syntax is required to access them.

Dictionary key:value pairs are not ordered, as you would expect. A list of the keys may be extracted using the keys() method, and values with the values() method.

You can also create dictionaries with just keys, from a list.

mydict = {}.fromkeys(mylist)

These can be used as look-up tables, or the values added later.

Dictionary keys can be any immutable type: strings, numbers, or tuples, but not mutable types, such as lists. To get a list of keys from an existing dictionary, then use the dictionary as a list:

keys = list(mydict)

Dictionary values

Objects stored can be of any type

- Lists, tuples, other dictionaries, etc...
- Can be accessed using multiple indexes or keys in `[ ]`
- Add a new value just by assigning to it

```
mydict = {'UK':['London', 'Wigan', 'Macclesfield', 'Bolton'],
          'US':['Miami', 'Springfield', 'New York', 'Boston']}
print(mydict['UK'][2])

homer = 1
print(mydict['US'][homer])

mydict['FR'] = ['Paris', 'Lyon', 'Bordeaux', 'Toulouse']
for country in mydict.keys():
    print(country, ': ', mydict[country])
```

```
FR :  ['Paris', 'Lyon', 'Bordeaux', 'Toulouse']
US :  ['Miami', 'Springfield', 'New York', 'Boston']
UK :  ['London', 'Wigan', 'Macclesfield', 'Bolton']
```

Shown is a simple dictionary containing just two keys ('UK' and 'US'), and each has a list as its value. The dictionary may be extended dynamically merely by assigning a value to a new key, 'FR' in the example. Notice how any original order of the keys is lost, since Python dictionaries are not ordered.
To access a "multi-dimensional" object, just add the key or index inside square brackets, as you would in most other languages. They can be literals (don't forget the quotes around keys) or variables.
We show a way of iterating through a dictionary, we will be discussing this in more detail later.

Removing items from a dictionary

To remove a single key/value pair:

- `del` *dict*`[`*key*`]`
- Raises a KeyError exception if the key does not exist
- *dict*`.pop(`*key*`[,`*default*`])`
- Returns *default* if the key does not exist

```
>>> fred={}
>>> del fred['dob']
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    del fred['dob']
KeyError: 'dob'
>>> fred.pop('dob', False)
False
```

- Also:
- *dict*`.popitem()` removes the next key/value pair used in iteration
- *dict*`.clear()` removes all key/value pairs from the dictionary

Deleting a key will delete the associated value as well. The value is not returned by the `del` statement, but is returned by the `pop()` method. Both can raise a `KeyError` exception if the key does not exist, but `pop()` can take an optional default value which is returned instead.
The `popitem()` method removes an arbitrary key/value pair, in that the order of keys within a dictionary is not defined. It would be of use if we were iterating through a dictionary removing each key/value in turn. `popitem()` returns the key/value pair deleted as a tuple, or raises a `KeyError` exception if the dictionary is empty.

Dictionary methods

dict.clear()	Remove all items from *dict*
dict.copy()	Return a copy of *dict*
dict.fromkeys(*seq*[,*value*])	Create a new dictionary from *seq*
dict.get(*key*[,*def*ault])	Return the value for *key*, or *default* if it does not exist
dict.items()	Return a view of the key-value pairs
dict.keys()	Return a view of the keys
dict.pop(*key*[,*default*])	Remove and return *key's* value, else return *default*
dict.popitem()	Remove the next item from the dictionary
dict.setdefault(*key*[,*default*])	Add *key* if it does not already exist
dict.update(*dictionary*)	Merge another dictionary into *dict*.
dict.values()	Return a view of the values

Return values from `keys()`, `values()`, and `items()` are *view objects*. These are discussed on the next slide.
Note that `copy()` returns a *shallow* copy of the dictionary, not a deep copy. See the *Advanced Collections* chapter for more on shallow and deep copies.

As we have said, Python dictionaries are **unordered**. However, this *might* be changing. Associated with other internal changes, in the C implementation of Python 3.6 the keys stay the order in which they were defined. To quote the documentation: "*The order-preserving aspect of this new implementation is considered an implementation detail and should not be relied upon*".
It is possible that this could become a language feature in the future, but until then, if you really need an ordered dictionary then it is safer to use `OrderedDict` from the **collections** module in the standard library.

View objects - examples

- May be used in iteration

```
nebula = {'M42':'Orion',
          'C33':'Veil',
          'M8' :'Lagoon',
          'M17':'Swan'
}
for kv in nebula.items():
    print(kv)
```

```
('M42', 'Orion')
('M17', 'Swan')
('M8', 'Lagoon')
('C33', 'Veil')
```

py3

- To store as a list

```
lkeys = list(nebula.keys())
print(lkeys)
```

```
['M42', 'M17', 'M8', 'C33']
```

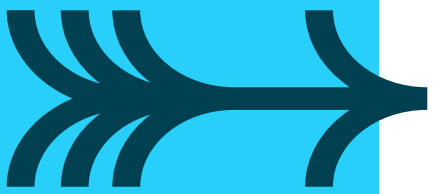
- In set operations

```
jelly = nebula.keys() | {'M37', 'M5'}
print(jelly)
```

```
{'M5', 'M37', 'M17', 'M42', 'M8', 'C33'}
```

View objects, as returned by **`items()`**, **`keys()`**, and **`values()`** methods, can be used in iteration and as objects to construct a list. The Set operations, `&`, `|`, `^` and –, may be used with a set on **`dict_keys`** and **`dict_items`** view objects, but not **`dict_values`**. Dictionary view objects are new to Python 3, but have also been introduced into 2.7.

SUMMARY



- **Lists are like arrays in other languages**
- **Tuples are "immutable"**
- But can contain variables
- **Slice lists and tuples using *object*[*start*:*end*+1]**
- **Sets store unordered unique objects**
- May be joined, along with other operations
- **Dictionaries store objects accessed by key**
- Keys are unique
- Not ordered