Python CheatSheet (For Linux)
Beginning Python: From Novice to Professional
Chapter 4(Dictionaries)

A data structure in which you can refer to each value by name. This type of structure is called a mapping. The only built-in mapping type in Python is the **dictionary**. The values in a dictionary don't have any particular order but are stored under a key, which may be a number, a string, or even a tuple.

Dictionay Uses

A dictionary is more appropriate than a list in many situations. Here are some examples of uses of Python dictionaries:

• Representing the state of a game board, with each key being a tuple of coordinates.
• Storing file modification times, with file names as keys.
• A digital telephone/address book.

Creating and Using Dictionaries

Dictionaries are written like this:

```
phonebook = {'Alice': '2341', 'Beth': '9102', 'Cecil': '3258'}
```

Dictionaries consist of **pairs (called items) of keys and their corresponding values**. In this example, the names are the keys and the telephone numbers are the values. Each key is separated from its value by **a colon (:)**, the items are separated by **commas**, and the whole thing is enclosed in **curly braces**. An empty dictionary (without any items) is written with just two curly braces, like this: {}.

Keys are unique within a dictionary (and any other kind of mapping). Values do not need to be unique within a dictionary.

The dict Function

You can use the dict function 1 to construct dictionaries from other mappings (for example, other dictionaries) or from sequences of (key, value) pairs:

```
>>> items = [('name', 'Gumby'), ('age', 42)]
>>> d = dict(items)
>>> d
{'age': 42, 'name': 'Gumby'}
>>> d['name']
'Gumby'
```

It can also be used with keyword arguments, as follows:

```
>>> d = dict(name='Gumby', age=42)
>>> d
{'age': 42, 'name': 'Gumby'}
```

Although this is probably the most useful application of dict, you can also use it with a mapping argument to create a dictionary with the same items as the mapping.

Basic Dictionary Operations

The basic behavior of a dictionary in many ways mirrors that of a sequence:

• **len(d)** returns the number of items (key-value pairs) in d.
• **d[k]** returns the value associated with the key k.
• **d[k] = v** associates the value v with the key k.
• **del d[k]** deletes the item with key k.
• **k in d** checks whether there is an item in d that has the key k.

Although dictionaries and lists share several common characteristics, there are some important distinctions:

Key types: Dictionary keys don't have to be integers (though they may be). They may be any immutable type, such as floating-point (real) numbers, strings, or tuples.

Automatic addition: You can assign a value to a key, even if that key isn't in the dictionary to begin with; in that case, a new item will be created. You cannot assign a value to an index outside

the list’s range (without using append or something like that).
Membership: The expression k in d (where d is a dictionary) looks for a key, not a value. The expression v in l, on the other hand (where l is a list) looks for a value, not an index.
Checking for key membership in a dictionary is much more efficient than checking for membership in a list. The difference is greater the larger the data structures are.

Dictionary Methods

clear

The clear method removes all items from the dictionary. This is an in-place operation (like list.sort), so it returns nothing.

```
>>> d = {}
>>> d['name'] = 'Gumby'
>>> d['age'] = 42
>>> d
{'age': 42, 'name': 'Gumby'}
>>> returned_value = d.clear()
>>> d
{}
>>> print returned_value
None
```

copy

The copy method returns a new dictionary with the same key-value pairs (a shallow copy, since the values themselves are the same, not copies):

```
>>> x = {'username': 'admin', 'machines': ['foo', 'bar', 'baz']}
>>> y = x.copy() (*** this is shallow copy, where changes affect x)
>>>z = deepcopy(x) (*** this is deep copy, where changes do not affect x )
>>> y['username'] = 'mlh'
>>> y['machines'].remove('bar')
>>> y
{'username': 'mlh', 'machines': ['foo', 'baz']}
>>> x
{'username': 'admin', 'machines': ['foo', 'baz']}
>>> xz
{'username': 'admin', 'machines': ['foo', 'bar', 'baz']}
```

fromkeys

The fromkeys method creates a new dictionary with the given keys, each with a default corresponding value of None:

```
>>> {}.fromkeys(['name', 'age'])
{'age': None, 'name': None}
```

get

The get method is a forgiving way of accessing dictionary items. Ordinarily, when you try to access an item that is not present in the dictionary, things go very wrong:

```
>>> d = {}
>>> print d['name']
Traceback (most recent call last):
File "<stdin>", line 1, in ?
KeyError: 'name'
Not so with get:
>>> print d.get('name')
None
```

has_key

The has_key method checks whether a dictionary has a given key. The expression d.has_key(k) is equivalent to k in d.

```
>>> d = {}
>>> d.has_key('name')
False
>>> d['name'] = 'Eric'
>>> d.has_key('name')
True
```

items and iteritems

The items method returns all the items of the dictionary as a list of items in which each item is of the form (key, value). The items are not returned in any particular order:

```
>>> d = {'title': 'Python Web Site', 'url': 'http://www.python.org', 'spam': 0}
>>> d.items()
[('url', 'http://www.python.org'), ('spam', 0), ('title', 'Python Web Site')]
```

The iteritems method works in much the same way, but returns an iterator instead of a list:

```
>>> it = d.iteritems()
>>> it
<dictionary-iterator object at 169050>
>>> list(it) # Convert the iterator to a list
[('url', 'http://www.python.org'), ('spam', 0), ('title', 'Python Web Site')]
```

Using iteritems may be more efficient in many cases (especially if you want to iterate over the result).

keys and iterkeys

The keys method returns a list of the keys in the dictionary, while iterkeys returns an iterator over the keys.

pop

The pop method can be used to get the value corresponding to a given key, and then remove the key-value pair from the dictionary:

```
>>> d = {'x': 1, 'y': 2}
>>> d.pop('x')
1
>>> d
{'y': 2}
```

popitem

The popitem method is similar to list.pop, which pops off the last element of a list. Unlike list.pop, however, popitem pops off an arbitrary item because dictionaries don't have a "last element" or any order whatsoever. This may be very useful if you want to remove and process the items one by one in an efficient way (without retrieving a list of the keys first):

```
>>> d
{'url': 'http://www.python.org', 'spam': 0, 'title': 'Python Web Site'}
>>> d.popitem()
('url', 'http://www.python.org')
>>> d
{'spam': 0, 'title': 'Python Web Site'}
```

Although popitem is similar to the list method pop, there is no dictionary equivalent of append (which adds an element to the end of a list). Because dictionaries have no order, such a method wouldn't make any sense.

setdefault

The setdefault method is somewhat similar to get, in that it retrieves a value associated with a given key. In addition to the get functionality, setdefault sets the value corresponding to the given key if it is not already in the dictionary:

```
>>> d = {}
>>> d.setdefault('name', 'N/A')
'N/A'
>>> d
{'name': 'N/A'}
>>> d['name'] = 'Gumby'
>>> d.setdefault('name', 'N/A')
'Gumby'
>>> d
{'name': 'Gumby'}
```

update

The update method updates one dictionary with the items of another:

```
>>> d = {
'title': 'Python Web Site',
'url': 'http://www.python.org',
'changed': 'Mar 14 22:09:15 MET 2008'
}
>>> x = {'title': 'Python Language Website'}
>>> d.update(x)
>>> d
{'url': 'http://www.python.org', 'changed':
'Mar 14 22:09:15 MET 2008', 'title': 'Python Language Website'}
```

The items in the supplied dictionary are added to the old one, supplanting any items there with the same keys.

The update method can be called in the same way as the dict function (or type constructor), as discussed earlier in this chapter. This means that update can be called with a mapping, a sequence (or other iterable object) of (key, value) pairs, or keyword arguments.

values and itervalues

The values method returns a list of the values in the dictionary (and itervalues returns an iterator of the values). Unlike keys, the list returned by values may contain duplicates:

```
>>>d = {}
>>>d[1] = 1
>>>d[2] = 2
>>>d[3] = 3
>>>d[4] = 1
>>>d.values()
[1, 2, 3, 1]
```