Chapter 18
Dictionaries

Abstract In addition to integers, floats, strings, and lists, Python has a powerful built-in Python data type called a dictionary. Dictionaries are useful for storing tables of information with a unique identifier for each record. Dictionaries provide a mapping from a set of keys to a set of values. In other words, given a key, a dictionary can look up the value associated with it. These have many applications in GIS, including reading GIS attribute tables or text data files and modifying them within a script. Also, they are often used to store pairs of items that go together. For example, soil science uses standard classifications for soil, abbreviated with terms such as 'Ap' and 'Cg'; However, more explicit names such as 'Plinthic Acrisol' (for 'Ap') and 'Gleyic Chernozem' (for 'Cg') are needed for some analysis. A dictionary can be used to store these terms so that the abbreviations are associated with the full names. This chapter shows the dictionary syntax for creating associations like this and then it shows how to access, update, and modify dictionaries.

Chapter Objectives
After reading this chapter, you'll be able to do the following:

- Create Python dictionaries.
- Add an item to a dictionary.
- Modify or delete an item from a dictionary.
- Check if a dictionary has a key.
- Explain `KeyError` exceptions.
- Replace `IF/ELSE IF` structures with dictionaries.
- List the keys, values, and items in a dictionary.
- Loop through a dictionary.
- Populate a dictionary based on user input.
- Populate a dictionary using `arcpy` cursors.
- Embed Python lists as values.

L. Tateosian, *Python For ArcGIS*, DOI 10.1007/978-3-319-18398-5_18

18.1 Dictionary Terms and Syntax

The terms used to refer to the elements in a Python dictionary are 'items', 'keys', and 'values'. Python dictionaries store a collection of *items;* Each item consists of a *key* and a *value*. An item is like a GIS attribute table record. Each record has a unique identifying attribute, such as its `FID` value, which allows you to access the rest of the values in that record. A Python dictionary item is a unique identifier, a key, along with the related value. The following code creates a Python dictionary containing five items:

```
>>> zipcodeDictionary = {27522 : 'Granville', 28736 : 'Jackson',
27953: 'Dare', 27511: 'Wake', 27607: 'Wake'}
```

The items in the dictionary are zipcode-county pairs; Each key is a North Carolina zip code and each value is the enclosing county name. Keys must be unique, but values can be duplicated. For example, two zip codes fall within Wake County. This is consistent with English language dictionaries; The words defined in an English dictionary must have only one entry, but synonyms have the same definition.

Dictionary keys are most often strings or numbers. Mutable types such as lists or dictionaries can not be used as keys. However, the item value can be of any data type, including a mixture of numbers, strings, lists, and so forth:

```
>>> flickDict = {8.5:1963, 'oceans':[11,12], 'up' : 'dog'}
```

The definition of a dictionary can be distributed across multiple lines. When multiple lines are used to assign a dictionary in the Interactive Window, three dots appear at the beginning of the lines:

```
>>> newVocab = {'mouse potato': 'a frequent computer user',
... 'wasband' : 'former husband',
... 'himbo': 'an attractive but vacuous man'}
```

The triple dots appear automatically if the dictionary extends across multiple lines in the Interactive Window to indicate that the lines are grouped together until the closing curly brace.

Python reports the data type of Python dictionaries as type `'dict'`:

```
>>> type(newVocab)
<type 'dict'>
```

Dictionaries can have zero or more items. The syntax for an assignment statement for a dictionary that has three items looks like this:

```
dictionaryName = {key1 : value1, key2 : value2, key3 : value3}
```

Spacing doesn't matter, but the punctuation is required. Each key is followed by a colon and then its corresponding value. The pairs are separated by commas. The curly braces indicate that it is a dictionary, just as square braces are used to create lists. Also similar to lists is the syntax for creating an empty dictionary. If dictionary items are not known ahead of time, you can create an empty dictionary and then add items to it. To create an empty dictionary, set a variable equal to a pair of empty curly braces:

```
dictionaryName = {} # Create an empty dictionary
```

18.1.1 Access by Key, Not by Index

The similarities between list and dictionary syntax can be misleading. Dictionaries store sequences of items, so you might assume you can try to index into a dictionary like a list or a string. But dictionaries are not designed to work that way. Instead, values need to be accessed using keys. The following code assigns a dictionary of student ids and dormitory room numbers (`'emforste'` and `'pgwodeho'` are roommates):

```
>>> roomDict = {'cslewis': 4139, 'emforste': 4118,
... 'jkrowlin': 4098, 'jrtolkie': 4259, 'pgwodeho': 4118,
... 'jrtolkie': 4259, 'vmhugo': 2121, 'vwoolf': 3145}
```

The following code uses the key, `'emforste'`, to retrieve this student's room number:

```
>>> roomDict['emforste']
4118
```

The syntax format for accessing an item is as follows:

```
theValue = dictionaryName[key1] # Get value paired with key1
```

The syntax looks similar to indexing into a list, but the numeric index is replaced by a key. Attempting to use zero-based indexing to access the second item in the dictionary throws an exception:

```
>>> roomDict[1]
Traceback (most recent call last):
File '<interactive input>', line 1, in <module>
KeyError: 1
```

Since `roomDict` is a dictionary, the number (1) is interpreted as a key. But the dictionary has no key of 1, so it reports a `KeyError`, an exception that is thrown whenever a key is not found in the set of existing keys.

18.1.2 Conditional Construct vs. Dictionary

A dictionary can sometimes be used to replace a multi-way conditional construct, one with multiple serially checked conditions. When a selection statement is controlling flow based on the value of some variable and actions taken inside each branch are based on the value of a second variable, the pairing of these values can be replaced with a dictionary. Some programming languages use a switch statement for this purpose; Python uses dictionaries instead. Take the following code as an example:

```
if num == 0:
    day = 'Monday'
    print 'Weekday: {0}'.format(day)
elif num == 1:
    day = 'Tuesday'
    print 'Weekday: {0}'.format(day)
elif num == 2:
    day = 'Wednesday'
    print 'Weekday: {0}'.format(day)
elif num == 3:
    day = 'Thursday'
    print 'Weekday: {0}'.format(day)
elif num == 4:
    day = 'Friday'
    print 'Weekday: {0}'.format(day)
```

The code prints the day of the week based on the value of `num`. The flow is controlled by the `num` variable and the code inside each branch is identical, except for the value of the `day` variable. `num` and `day` are paired by the branching. As a sleeker alternative, a dictionary can create this mapping from `num` values to `day` values, as in Example 18.1.

Example 18.1

```
# weekdays.py
weekdayDict = {0: 'Monday', 1: 'Tuesday', 2: 'Wednesday',
               3: 'Thursday', 4: 'Friday'}
day = weekdayDict[num]
print 'Weekday: {0}'.format(day)
```

These four lines of code replace the entire multi-way conditional construct. Accessing the dictionary with `num` as the key, returns the value of `day` used in the print statement. The key access statement replaces the hard-coded weekday name assignment statements in each block of the conditional construct.

18.1.3 How to Modify: Update/Add/Delete Items

Dictionaries can be modified by updating existing items, adding new items, and deleting items. Each of these three actions access the dictionary using a key in square braces. Modifying mutable and immutable item values is handled slightly differently. When the item values are immutable data types like strings or numbers, the syntax for modifying an item and adding a new item is identical:

```
# update or add an item with an immutable value
dictionaryName[ key ] = value
```

If the key exists in the dictionary, the item is modified. If not, a new item is added. As an example of modifying an existing item, suppose `'emforste'` and `'pgwodeho'` had a falling out, causing `'emforste'` to move into room 4139 with `'cslewis'`. The following code updates an existing item, modifying the entry for `'emforste'` with an assignment statement to change the value associated with key `'emforste'`:

```
>>> # Modify the room number for 'emforste'.
>>> roomDict['emforste'] = 4139
>>> roomDict # The dictionary is updated.
{'emforste': 4139, 'cslewis': 4139, 'jrtolkie': 4259,
'vmhugo': 2121, 'vwoolf': 3145, 'pgwodeho': 4118,
'jkrowlin': 4098}
>>> len(roomDict)
7
```

Because the student, `'emforste'` was already listed in the dictionary, this code modified that item. The number of items in the dictionary remains 7. However, if a new student, 'lcarroll' moves into the dorm, this is not an existing key. The following code adds a new item and assigns 'lcarroll' to room 2121 with 'vmhugo':

```
>>> # Add 'lcarroll' in room 2121 to the dictionary.
>>> roomDict['lcarroll'] = 2121
>>> roomDict # A new item is added to the dictionary.
{'emforste': 4139, 'cslewis': 4139, 'jrtolkie': 4259,
'lcarroll': 2121, 'vmhugo': 2121, 'vwoolf': 3145,
'pgwodeho': 4118, 'jkrowlin': 4098}
>>> len(roomDict) # The length of the dictionary becomes 8.
8
```

When the item values are mutable data types, they may need to be updated differently. Consider the following dictionary containing two weather gauge sites and a list of temperature readings at each site.

```
>>> weather = {'S1' : [15,20],'S2' : [25,30,40]}
```

Adding another site to the dictionary or replacing one of the items, has the same format as the numerical and string examples. The value for new site 'S3' is a list, so the statement has two sets of square braces, one around the key and one around the value (list) being assigned on the right:

```
>>> weather['S3'] = [33,40]
>>> weather
{'S3': [33, 40], 'S2': [25, 30, 40], 'S1': [15, 20]}
```

Other operations, such as adding another temperature reading to a site list or incrementing each temperature in a site list, require techniques that are specific to list handling. To add a temperature to an existing item list, you must use the append method on that item. No assignment statement is used with the append method since it is an in-place method that modifies the mutable object. For example, the following statement adds 36 to the 'S2' temperature list:

```
>>> weather['S2'].append(36)
>>> weather
{'S3': [33, 40], 'S2': [25, 30, 40, 36], 'S1': [15, 20]}
```

A list comprehension can be used to increment each temperature in a site list. The following code uses list comprehension to creates a list of incremented temperatures and then assigns the new list to site 'S1':

```
>>> update = [i + 1 for i in weather['S1']]
>>> weather['S1'] = update
>>> weather
{'S3': [33, 40], 'S2': [25, 27, 40, 36], 'S1': [16, 21]}
```

Deleting a dictionary item works the same for both mutable and immutable values. Access the item with its key and use the Python keyword `del` with the following syntax:

```
# Delete the item with the key named key1
del dictionaryName[key1]
```

Returning to the dormitory room example, suppose `'jrtolkie'` moves out of the dormitory so that he can focus on his next novel, tentatively titled 'Everybody loves rings'. The following code deletes the item with key `'jrtolkie'`, reducing the number of items to 7:

```
>>> del roomDict['jrtolkie']
>>> roomDict
{'emforste': 4139, 'cslewis': 4139, 'lcarroll': 2121, 'vmhugo':
2121, 'vwoolf': 3145, 'pgwodeho': 4118, 'jkrowlin': 4098}
>>> len(roomDict)
7
```

If the key used in a deletion statement is missing, a `KeyError` exception is thrown:

```
>>> del roomDict['cslouis']
Traceback (most recent call last):
File '<interactive input>', line 1, in <module>
KeyError: 'cslouis'
```

18.2 Dictionary Operations and Methods

Like strings and lists, dictionaries have specialized operation and methods for dictionary operations. This section focuses on four frequently used operations and methods: the `in` keyword, `keys`, `values`, and `items`.

18.2.1 Does It Have That Key?

The `in` method can check if a dictionary has a particular key to avoid overwriting an existing key or to avoid throwing a `KeyError`. The `in` keyword can be used on keys in a manner similar to how it works on lists. It returns `True` or `False`. To check for the keys 'cslewis' and 'cslouis' in the dormitory room dictionary, use the following code:

```
>>> 'cslewis' in roomDict
True
>>> 'cslouis' in roomDict
False
```

The `in` keyword can be used to avoid `KeyError` exceptions that occur when a statement attempts to access or delete an item with a nonexistent key:

```
>>> key = 'cslouis'
>>> if key in roomDict
...     print roomDict[key]
... else:
...     'No {0} key found.'.format(key)
...
'No cslouis key found.'
```

18.2.2 Listing Keys, Values, and Items

Like other strings and lists, dictionaries have specialized methods for dictionary operations. Certain dictionary methods allow you to list the keys, values, and items. `keys` returns a list of the keys in the dictionary in arbitrary order:

```
>>> roomDict.keys()
['emforste', 'cslewis', 'lcarroll', 'vmhugo', 'vwoolf', 'pgwodeho',
'jkrowlin']
```

`values` returns a list of the values in the dictionary:

```
>>> roomDict.values()
[4139, 4139, 2121, 2121, 3145, 4118, 4098]
```

`items` returns a list of the items in the dictionary as key-value tuples:

```
>>> roomDict.items()
[('emforste', 4139), ('cslewis', 4139), ('lcarroll', 2121), ('vmhugo', 2121),
('vwoolf', 3145), ('pgwodeho', 4118), ('jkrowlin', 4098)]
```

18.2.3 Looping Through Dictionaries

The listing methods can be used to loop through a dictionary. Typically, you won't need to store the lists returned by these methods, since they are already stored in the dictionary; In this case, you can just use the method call as the sequence that comes after the `in` keyword in a `FOR`-loop. For the `keys` and `values` method, place a single variable between the `for` and `in` keywords. The variable iterates through the keys or values:

```
>>> for k in newVocab.keys():
...     print k
...
himbo
wasband
mouse potato
```

```
>>> for v in newVocab.values():
...     print v
...
an attractive but vacuous man
former husband
a frequent computer user
```

For the `items` method, place two variables between the `for` and `in` keywords. One iterates through the keys and the other one iterates through the values:

```
>>> for k, v in newVocab.items():
...     print k, ':', v
...
himbo : an attractive but vacuous man
wasband : former husband
mouse potato : a frequent computer user
```

The `keys` method can be used to update each item's value. The following example uses the `keys` method to append zero to every weather gauge temperature list:

```
>>> for k in weather.keys():
...     weather[k].append(0)
...
>>> weather
{'S3': [33, 40, 0], 'S2': [25, 27, 40, 36, 0], 'S1': [16, 21, 0]}
```

The `values` method can be used to derive information from each value. The following example uses the `values` method to find the average temperature at each site:

```
>>> for v in weather.values():
...     print sum(v)/len(v),
...
24 25 12
```

The `items` method is useful when both key and value are needed in the loop. The following example uses the `items` method to print the fourth floor residents in the dormitory room dictionary:

```
>>> for k, v in roomDict.items():
...     room = str(v)
...     if room.startswith('4'):
...         print '{0}, room {1}'.format(k,v)
...
emforste, room 4139
cslewis, room 4139
pgwodeho, room 4118
jkrowlin, room 4098
```

18.3 Populating a Dictionary

The examples thus far have hard-coded dictionaries to introduce dictionary syntax, but you'll often want to generate dictionaries dynamically. This section shows how to populate a dictionary to store data from user input. As a starting point, the first examples use a hard-coded set of keys and the built-in `raw_input` function to gather values. The remaining examples generate the entire dictionary dynamically, using directory listings or `arcpy` cursors to retrieve data from attribute tables.

The script 'healthyLiving.py' in Example 18.2 surveys user preferences. The script starts with an empty dictionary and uses an assignment statement to add new values. An empty dictionary `favDict` is created, then the script loops through the topics, asks about each topic in turn and populates the dictionary inside the loop.

Example 18.2

```
# healthyLiving.py
# Purpose: Collect user preferences; only keep most recent responses.
topics = ['fruit','fruit','fruit',
          'veg','veg','veg','exercise','park']
favDict = {}    # Create empty dictionary
for topic in topics:
    question = 'What is your favorite {0}?'.format(topic)
    answer = raw_input(question)
    favDict[topic] = answer    # Add or update item
print favDict
```

The following assignment statement from Example 18.2 populates the dictionary:

```
favDict[topic] = answer # Add or update item
```

The script doesn't check if the dictionary has a key already before assigning a value. The scripts asks for a favorite fruit and vegetable more than once. When the user responses are `grape`, `banana`, `mango`, `fennel`, `lettuce`, `carrots`, `rollerblading`, and `Central` in that order, the dictionary records `grape` but overwrites it when `banana` is given. In the end, the dictionary shows `mango` as a favorite, since this was the latest favorite fruit response:

```
>>> favDict
{'veg': 'carrots', 'fruit': 'mango', 'park': 'Central', 'exercise':
'rollerblading'}
```

By using only the latest response, Example 18.2 assigns only one answer to each question. This overwriting could be prevented by checking for the key. Replacing the loop in Example 18.2 with the following code would record only first responses and discard all other answers:

```
for topic in topics:
    question = 'What is your favorite {0}?'.format(topic)
    answer = raw_input(question)
    if key not in roomDict:
        favDict[topic] = answer   # Add item (only 1st responses recorded)
```

The script 'healthLivingV2.py', in Example 18.3 takes an alternative approach that preserves multiple responses by using Python lists as item values.

Example 18.3

```
 # healthyLivingV2.py
 # Purpose: Collect user preferences; keep all responses.
 topicList = ['fruit','fruit','fruit',
              'veg','veg','veg','exercise','park']
 topDict = {}    # Create empty dictionary
 for topic in choiceList:
     question = "What is your favorite {0}?".format(topic)
     answer = raw_input(question)
     if topic not in topDict:
         # Add a new item to the dictionary.
         topDict[topic] = [answer]
     else:
         # Update an item by adding to an item's list.
         topDict[topic].append(answer)
print 'topDict {0}'.format(topDict)
```

The `topDict` dictionary includes every answer, listing it with the corresponding question:

```
>>> topDict
{'veg': ['fennel', 'lettuce', 'carrots'], 'fruit': ['grape', 'banana',
'mango'], 'park': ['Central'], 'exercise': ['rollerblading']}
```

Each key has exactly one value, a list. But we are storing one or more answer for each question. For example, `topDict` includes three favorite vegetables and three favorite fruits. The script collects the list type values by using decision-making blocks. If the dictionary does not yet have a topic as a key, it adds a new item to the dictionary. The values need to be lists, so that more than one answer can be associated with each question. The value of the new item is the current answer surrounded by square braces. In other words, the new item is a list with one element, the current answer. The ELSE block appends the new answer to the existing list for that topic. Table 18.1 juxtaposes the pseudocode for Example 18.2 (collecting string values) and Example 18.3 (collecting list values).

Variations of the pseudocode in Table 18.1 can be used in many applications. As another example of using dictionaries to collect information, consider 'fileDates.py' in Example 18.4, which uses a dictionary to collect file names and modification time stamps. The script lists the files in a given directory and stores a file name in a dictionary along with a modification time-stamp. Key conflicts might arise if the script were to walk into subdirectories. But since this script only looks in one directory, there is no chance of a duplicate file name causing a dictionary item to be overwritten. Therefore, the dictionary population code was modeled after P1 in Table 18.1.

Table 18.1 Pseudocode for Examples 18.1 and 18.2 collecting data in a dictionary.

	Dictionary population pseudocode	**Add and append syntax**
P1	`# String values`	
	`CREATE empty dictionary, D`	
	`GET list, L`	
	`FOR each v_i in L`	
	`  GET v_j`	
	`  ADD to new item to D`	`D[v_i] = v_j`
	`    with key of v_i & value of v_j`	
	`ENDFOR`	
P2	`# List values`	
	`CREATE empty dictionary, D`	
	`GET list, L`	
	`FOR each v_i in L`	
	`  GET v_j`	
	`  IF D does not have key v_i THEN`	
	`    ADD new item to D with key of v_i`	`D[v_i] = [v_j]`
	`      and value of a list`	
	`      containing v_j`	
	`  ELSE`	`D[v_i].append(v_j)`
	`    APPEND v_j to D item with key v_i`	
	`  ENDIF`	
	`ENDFOR`	

`fileDict` contains file name-time pairs. The first few items in the dictionary from Example 18.4 look something like this following:

```
{'a.shx': 'Thu Jan 01 13:34:24 2009', 'a.dbf': 'Thu Jan 01 13:34:24
2009', 'dog.JPG': 'Thu Jan 01 13:34:24 2009',...
```

Example 18.4

```
# fileDates.py
# Purpose: Collect filenames and modification dates in a dictionary.
import os, time
inputDir = 'C:/gispy/data/ch18/smallDir'

fileList = os.listdir(inputDir)
fileDict = {}
for f in fileList:
    epochNum = os.path.getmtime(inputDir + '/' + f)
    modTime = time.ctime(epochNum)
    fileDict[f] = modTime

print fileDict
```

18.3.1 Dictionaries and Cursors

Another common use of dictionaries is to store GIS data gathered by `arcpy` cursors. Example 18.5 finds the median area of the polygons in a shapefile and then determine the IDs of any polygons whose areas are close to the median area. The script collects the polygon areas of a shapefile in a dictionary. Then the built-in `numpy` module is used to calculate the median of the dictionary value list. A final loop through the dictionary items identifies the polygons with area measurements in the range of plus or minus 400 square feet of the median area.

Example 18.5

```
# areaMedian.py
import arcpy, numpy

fc = 'C:/gispy/data/ch18/parkAreas.shp'
idField = 'FID'
areaField = 'F_AREA'
areasDict = {}

# Populate dictionary with id:area items.
sc = arcpy.da.SearchCursor(fc, [idField, areaField])
for row in sc:
    fid = row[0]
    area = row[1]
    areasDict[fid] = area
del sc

# Find the median area.
areas = areasDict.values()
medianArea = numpy.median(areas)
print 'Median area: {0}'.format(medianArea)

# Find the polygons with values close to median.
sqft = 400
print 'Polygons close to median:'
for k, v in areasDict.items():
    if medianArea - sqft < v < medianArea + sqft:
        print '{0}: {1}, {2}: {3}'.format(idField, k, areaField, v)
```

Populating the dictionary using a cursor looks similar to the previous examples. The only difference is that, in this case, the values are collected by looping with a search cursor. In Example 18.5, no conditional constructs are used when the dictionary is populated because the keys are `FID` values, which are unique. The values are simply numbers, so Example 18.1 follows the Table 18.1 P1 pseudocode.

If you want to collect information about a field that has duplicate values, you may want to follow the Table 18.1 P2 pseudocode instead. Example 18.5 finds the median area of each land cover type. Multiple polygons have the same land cover types, so the script needs to collect a list of areas for each type. This script encounters three cover types (`'woods'`, `'other'`, and `'orch'`), so the final dictionary has three items, one key for each cover type and one list of areas for each cover type. The final loop finds the median of each area list in the dictionary. The output from Example 18.5 is as follows:

```
Polygons with cover 'woods' have median area 83095.3479504
Polygons with cover 'other' have median area 55491.6260843
Polygons with cover 'orch' have median area 83477.7527484
```

Example 18.6

```python
# coverMedianArea.py
import arcpy, numpy

arcpy.env.workspace = 'C:/gispy/data/ch18'
fc = 'parkAreas.shp'

# Populate the dictionary,
# accumulate a list of areas for each cover type.
d = {}
sc = arcpy.da.SearchCursor(fc, ['COVER', 'F_AREA'])
for row in sc:
    cover = row[0]
    area = row[1]
    if d.has_key(cover):
        d[cover].append(area)
    else:
        d[cover] = [area]
del sc

# Calculate the median area for each cover type.
for k, v in d.items():
    median = numpy.median(v)
    print '''Polygons with cover '{0}' have \
             median area {1}'''.format(k, median)
```

These examples used the `numpy` modules. `numpy` is a scientific computing module optimized for efficiently performing operations on n-dimensional arrays. See the online documentation and examples for more information on the `numpy` module.

18.4 Discussion

Dictionaries provide a way to store a mapping between one set of values and another set of values. Like strings, lists, and tuples, they are a built-in Python sequence data type that has methods and special operations. Items are accessed via a key, not by indexing. The `'dict'` dictionary data type is not ordered (though Python does have other variations of dictionaries that are ordered). A dictionary can be hard-coded or populated dynamically. The syntax for adding items uses an assignment statement with the dictionary name and key in square braces on the left and a value on the right. If the key is not already in the dictionary, a new item is added. If, on the other hand, the key is already in the dictionary, the item with this key is overwritten. Dictionaries with value lists are often useful. In this case, list syntax and methods must be used to modify the lists.

Common applications of lists include replacing multi-way conditional constructs (`if/elif/elif...`) and storing data collected from user input or with `arcpy` cursors. Care must be taken to use the `in` keyword as appropriate when populating a dictionary, since overwriting can occur. Handling these situations depends on the application. If the input already has unique keys (such as FIDs) no checking may be needed. But if collisions can occurs, thought should be given to how to handle these. Should the script keep the first instance of that key? Should it keep the last one? Should it use a list and store every occurrence? The examples in this chapter can be used to help with these decisions.

18.5 Key Terms

`'dict'` data type
Key
Value
Item
Access by key vs. indexing multi-way conditional construct
Dictionary methods (`keys, values, items`)
`numpy` module

18.6 Exercises

1. **landUse.py** Sample script, 'landUse.py' has a land usage dictionary `landUse = {'res': 1, 'com': 2, 'ind': 3, 'other' :[4,5,6,7]}`. Add code to the script to use common dictionary operations and methods to perform the following tasks:

   (a) Print the value of the item with key `'com'`.
   (b) Check if the dictionary has key `'res'` and print the result.
   (c) Increment the value of the item with key `'ind'` and then print the result.

(d) Use access by key format to add an item with land use of `'agr'` and a value of 0. (Then print the dictionary.)
(e) Change the land use `'res'` value to 10. (Then print the dictionary.)
(f) Print a list of the dictionary keys.
(g) Print the dictionary values, one per line, using a `FOR`-loop.
(h) Print a list of the dictionary items.
(i) Delete the item with key `'ind'`. (Then print the dictionary.)
(j) Check for membership of the key `'ind'`, using the keyword `in`. Print the result.

Printed output should appear as follows:

```
>>> 1. Print the value of the item with key 'com': 2
2. Check if the dictionary has key 'res': True
3. Increment the value of the item with key 'ind': 3
4. Add an item with land use 'agr' a value of 0:
   {'ind': 3, 'res': 1, 'other': [4, 5, 6, 7], 'com': 2,
   'agr': 0}
5. Change land use 'res' value to 10.
   {'ind': 3, 'res': 10, 'other': [4, 5, 6, 7], 'com': 2,
   'agr': 0}
6. Print a list of the dictionary keys:
   ['ind', 'res', 'other', 'com', 'agr']
7. Print the dictionary values:
3
10
[4, 5, 6, 7]
2
0
8. Print a list of the dictionary items:
   [('ind', 3), ('res', 10), ('other', [4, 5, 6, 7]), ('com',
   2), ('agr', 0)]
9. Delete the item with key 'ind':
   {'res': 10, 'other': [4, 5, 6, 7], 'com': 2, 'agr': 0}
10. Check for membership of the key 'ind': False
```

2. **tideRecords.py** Sample script, 'tideRecords.py' has a tide gauge dictionary `tides={'G1': [1,6], 'G2': [2], 'G3': [3,8,9]}`. The dictionary currently stores tide gauge readings for three gauges, `G1`, `G2`, and `G3`. Notice that each value in the dictionary is a Python list, so that one or more readings can be recorded for each gauge. Whenever a new item is added to the dictionary, the value must be a Python list, so that the item can store multiple readings for that gauge. Use dictionary operations to reflect each of the following events and print the dictionary after each step:

(a) A new gauge, `G5` has been installed and the first reading was `2`.
(b) Record an additional reading for gauge `G5`, the number `6`.
(c) The latest reading from gauge `G3` is invalid. Discard this information using the list `pop` method.

(d) A new gauge `G6` has been installed but no readings are recorded yet. Use an empty list as a placeholder.
(e) Gauge `G3` is no longer collecting data. Remove the item that represents this gauge.
(f) Gauge `G1` recorded measurements two times higher than they should be. Use list comprehension to correct this error.

Printed output should appear as follows:

```
>>> 1. {'G5': [2], 'G3': [3, 8, 9], 'G2': [2], 'G1': [1, 6]}
2. {'G5': [2, 6], 'G3': [3, 8, 9], 'G2': [2], 'G1': [1, 6]}
3. {'G5': [2, 6], 'G3': [3, 8], 'G2': [2], 'G1': [1, 6]}
4. {'G6': [], 'G5': [2, 6], 'G3': [3, 8], 'G2': [2], 'G1':
   [1, 6]}
5. {'G6': [], 'G5': [2, 6], 'G2': [2], 'G1': [1, 6]}
6. {'G6': [], 'G5': [2, 6], 'G2': [2], 'G1': [0.5, 3.0]}
```

3. **tideRecordsLoop.py** Sample script, 'tideRecordsLoop.py' has a dictionary `tides={'G1': [1,6], 'G2': [2], 'G3': [3,8,9]}`. These represent tide gauge readings for three gauges, `'G1'`, `'G2'`, and `'G3'`. Use the `keys`, `values`, and `items` dictionary methods to perform the following updates and calculations:

(a) Append a reading of 7 for each gauge (Then print the dictionary.)
(b) Print the number of readings for each gauge.
(c) Square the value of every reading for every gauge. (Then print the dictionary.)
(d) Find the minimum reading for each gauge in the dictionary.

Printed output should appear as follows:

```
>>> 1. {'G3': [3, 8, 9, 7], 'G2': [2, 7], 'G1': [1, 6, 7]}
2. Number of readings: 4
2. Number of readings: 2
2. Number of readings: 3
3. {'G3': [9, 64, 81, 49], 'G2': [4, 49], 'G1': [1, 36, 49]}
4. Min reading at gauge G3 = 9
4. Min reading at gauge G2 = 4
4. Min reading at gauge G1 = 1
```

4. **check4keys.py** The sample script 'check4keys.py' contains a dictionary. Add code to the script so that it takes 1 or more numerical arguments, checks if the dictionary has those numbers as keys, and reports the results as shown in the following example:

Example input: 22 11 50 75

Example output:

```
>>> Key 22.0 has value: 4
Key 11.0 not found.
Key 50.0 not found.
Key 75.0 has value: 8
```

5. **sizeDict.py** Given a directory path as an argument, collect the file names and file sizes in a dictionary and print the dictionary.

   Example input: C:/gispy/data/ch18/smallDir

   Example output:
   ```
   >>> Size dictionary:
   {'a.shx': 124L, 'a.dbf': 2540L, 'dog.JPG': 579880L, 'pines.JPG':
   588644L, 'a.shp': 620L, 'a.prj': 145L, 'a1.lyr': 10752L, 'cluck.
   wav': 33970L, 'a.kml': 4684L, 'xyData2.txt': 124L, 'a.shp.xml':
   1644L, 'abc.dbf': 250L}
   Average file size: 98466.0769231
   ```

6. **typeDict.py** Given a directory path as an argument, collect the file data types and names in a dictionary. The dictionary should have one item for each data type, as determined by the `arcpy Describe` object. Each item should have a list of files with the key data type. Print the resulting dictionary and compare it to the given example.

   Example input: C:/gispy/data/ch18/smallDir

   Example output:
   ```
   >>> Type dictionary:
   {u'Layer': ['a1.lyr'], u'DbaseTable': ['abc.dbf'], u'ShapeFile':
   ['a.dbf', 'a.shp'], u'File': ['a.kml', 'a.prj', 'a.shp.xml', 'a.
   shx','cluck.wav'], u'TextFile': ['xyData2.txt'], u'RasterDataset':
   ['dog.JPG', 'pines.JPG']}
   ```

7. **multiwayDict.py** The sample script 'multiwayDict.py' computes a list of values based on a given sea level rise (SLR) scenario and a time interval. It uses one branch for each of six SLR scenarios (B1, A1T, B2, A1B, A2, and A1F1). Each branch applies a rise rate that corresponds to the scenario. Rewrite this script to use the following dictionary of SLR scenarios and rates:

   ```
   rateDict = {'B1': 0.0038, 'A1T': 0.0045, 'B2': 0.0043, 'A1B':
   0.0048, 'A2': 0.0051, 'A1F1':0.0059}
   ```

   Remove the multi-way conditional construct. Replace the repeated code in each block with a single set of statements that uses the dictionary keys and values to achieve the same affect. Include code to check that the dictionary has the input key. The revised script should be less than 25 lines.

   Example input1: A1B 50

   Example output1:
   ```
   >>> Running Sea Level Rise Model for A1B [0.24, 0.48, 0.72]
   ```

Example input2: AZ 5

Example output2:

```
>>> Warning: Invalid resolution. Choose B1, A1T, B2, A1B, A2,
or A1F1.
```

8. **trees.py** Write a script that will read a dBASE file and collect information in a dictionary about two of the dBASE fields. The script should require three arguments, the full path name of a dBASE file, the name of a classification field in the dBASE file (e.g., species), and the name of a second field in the dBASE file (e.g., DBH). Create a dictionary with a key for each distinct classification. Use the value of the second field to set the dictionary item's value. Use only the *first occurrence* of a classification encountered by the cursor (e.g., record only the first `'LOB'`, which has a DBH of 24). Print the dictionary.

   Example input: C:/gispy/data/ch18/rdu_forest1.dbf SPECIES DBH

   Example output:

   ```
   >>> Tree dictionary:
   {u'LOB': 24, u'BE': 17, u'WD': 17, u'WO': 14, u'HK': 9, u'YP': 17,
   u'POP': 21, u'RB': 15, u'RM': 7, u'ASH': 11, u'LP': 23, u'SLP': 14,
   u'VP': 9, u'SRW': 5, u'SHL': 16, u'CV': 18, u'RO': 8, u'MPL':
   5, u'SP':13, u'SW': 5, u'MW': 4, u'SL': 11, u'SG': 14}
   ```

9. **trails.py** Practice using cursors with dictionaries by writing a script to use a dictionary to calculate some statistics for a shapefile. The script should require three arguments, the full path name of a shapefile, a unique identifying field name, and a numerical field. As an example, we'll use a shapefile containing trail widths for a park in Narnia. First, populate and print a dictionary to pair the input fields using the ID field for the keys and the numeric field as the value. The 'numeric values' in the sample data, the trail width field, are stored in a text type field, so they should be cast to floats before being inserted into the dictionary. Next, use the dictionary values to find and print the minimum and maximum values of the trail width field. Also, determine and print the dictionary keys whose trail width values are minimal and the keys whose trail width values are maximal as shown in the example (The example output only shows part of the dictionary).

   Example input: C:/gispy/data/ch18/narniaHike.shp FID Tra_Width

   Example output:

   ```
   >>> FID_Tra_Width_Dict = {{0: 12.0, 1: 30.0, 2: 24.0, 3: 18.0,...
   Minimum Tra_Width = 6.0
   Features(s) with minimum Tra_Width: 6 65 82 114
   Maximum Tra_Width = 48.0
   Features(s) with maximum Tra_Width: 56 164
   ```

10. **trailWidths.py** Write a script to use a dictionary and find the average trail width for each classification in the Narnia trail shapefile. The script should require three arguments, the full path name of a shapefile, a classification field name, and a trail width field. Populate and print a dictionary with classifications as keys and lists of trail widths as values. The trails widths in the sample data are stored as strings, so they must be cast to floats. Then find and print the average trail width for each classification as shown in the example (The example output only shows part of the dictionary).

    Example input: C:/gispy/data/ch18/narniaHike.shp Classifica Tra_Width

    Example output:

```
>>> Classifica_Tra_Width_Dict = {u'Barren': [18.0, 24.0, 30.0, 18.0,
18.0, 12.0, 30.0, 18.0, 24.0, 30.0, 18.0], u'Some Bare Ground':
[18.0, 12.0, 12.0,...
Classifica: Barren, average Tra_Width: 21.82
Classifica: Some Bare Ground, average Tra_Width: 20.68
Classifica: Stunted Vegetation, average Tra_Width: 18.39
```

Tip The following code rounds one number (Pi) to two decimal places and another number (e) to three decimal places:

```
>>> import math
>>> print 'The numbers {0:.2f} and {1:.3f}'.format(math.pi, math.e)
The numbers 3.14 and 2.718
```