6. Dictionaries, sets, files, and modules

openbookproject.net/books/bpp4awd/ch06.html

6.1. Dictionaries

Dictionaries are a compound type different from the sequence types we studied in the Strings, lists, and tuples chapter. They are Python's built-in **mapping type**. They map **keys**, which can be any immutable type, to values, which can be any type, just like the values of a list or tuple.

Note

Other names for dictionaries in computer science include *maps*, *symbol tables*, and associative arrays. The pairs of values are referred to as *name-value*, *key-value*, *field-value*, or attribute-value pairs.

As an example, we will create a dictionary to translate English words into Spanish. For this dictionary, the keys are strings.

One way to create a dictionary is to start with the empty dictionary and add **key-value pairs**. The empty dictionary is denoted with a pair of **curly braces**, `{}` :

```
>>> eng2sp = {}
>>> type(eng2sp)
<class 'dict'>
>>> eng2sp['one'] = 'uno'
>>> eng2sp['two'] = 'dos'
>>> eng2sp['three'] = 'tres'
```

The first assignment creates a dictionary named `eng2sp` ; the other assignments add new key-value pairs to the dictionary. We can print the current value of the dictionary in the usual way:

```
>>> print(eng2sp)
{'three': 'tres', 'one': 'uno', 'two': 'dos'}
```

The key-value pairs of the dictionary are seperated by commas. Each pair contains a key and a value separated by a colon.

The order of the pairs may not be what you expected. Python uses complex algorithms to determine where the key-value pairs are stored in a dictionary. For our purposes we can think of this ordering as unpredicatable, so you should not try to rely on it. Instead, look up values by using a known key.

Another way to create a dictionary is to provide a list of key-value pairs using the same syntax as the previous output:

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

It doesn't matter what order we write the pairs. The values in a dictionary are accessed with keys, not with indices, so ordering is unimportant.

Here is how we use a key to look up the corresponding value:

```
>>> eng2sp['two']
'dos'
```

The key `'two'` yields the value `'dos'` .

6.2. Dictionary operations

The `del` statement removes a key-value pair from a dictionary. For example, the following dictionary contains the names of various fruits and the number of each fruit in stock:

```
>>> inventory = {'apples': 430, 'bananas': 312, 'oranges': 525, 'pears': 217}
>>> print(inventory)
{'apples': 430, 'bananas': 312, 'pears': 217, 'oranges': 525}
```

If someone buys all of the pears, we can remove the entry from the dictionary:

```
>>> del inventory['pears']
>>> print(inventory)
{'apples': 430, 'bananas': 312, 'oranges': 525}
```

Or if we're expecting more pears soon, we might just change the value associated with pears:

```
>>> inventory['pears'] = 0
>>> print(inventory)
{'apples': 430, 'bananas': 312, 'pears': 0, 'oranges': 525}
```

The `len` function also works on dictionaries; it returns the number of key-value pairs:

```
>>> len(inventory)
4
```

The `in` operator returns `True` if the key appears in the dictionary and `False` otherwise:

```
>>> 'pears' in inventory
True
>>> 'blueberries' in inventory
False
```

This operator can be very useful, since looking up a non-existant key in a dictionary causes a runtime error:

```
>>> inventory['blueberries']
Traceback (most recent call last):
  File "", line 1, in <module>
KeyError: 'blueberries'
>>>
```

To address this problem, the built-in `get` method provides a default value that is returned when a key is not found:

```
>>> inventory.get('blueberries', 0)
0
>>> inventory.get('bananas', 0)
312
```

Python's built-in `sorted` function returns a list of a dictionaries keys in sorted order:

```
>>> sorted(inventory)
['apples', 'bananas', 'oranges', 'pears']
```

6.3. Aliasing and copying

Because dictionaries are mutable, you need to be aware of aliasing. Whenever two variables refer to the same object, changes to one affect the other.

If you want to modify a dictionary and keep a copy of the original, use the `copy` method. For example, `opposites` is a dictionary that contains pairs of opposites:

```
>>> opposites = {'up': 'down', 'right': 'wrong', 'true': 'false'}
>>> an_alias = opposites
>>> a_copy = opposites.copy()
```

`an_alias` and `opposites` refer to the same object; `a_copy` refers to a fresh copy of the same dictionary. If we modify `alias`, `opposites` is also changed:

```
>>> an_alias['right'] = 'left'
>>> opposites['right']
'left'
```

If we modify `a_copy`, `opposites` is unchanged:

```
>>> a_copy['right'] = 'privilege'
>>> opposites['right']
'left'
```

6.4. Sets

A **set** is a Python data type that holds an unordered collection of unique elements. It implements the set abstract data type which is in turn based on the mathematical concept of a finite set. As with dictionaries, Python uses curly braces to indicate a set, but with elements instead of key-value pairs:

```
>>> what_am_i = {'apples': 32, 'bananas': 47, 'pears': 17}
>>> type(what_am_i)
<class 'dict'>
>>> what_am_i = {'apples', 'bananas', 'pears'}
>>> type(what_am_i)
<class 'set'>
```

To create an empty set, you can not use empty curly braces.

```
>>> what_am_i = {}
>>> type(what_am_i)
<class 'dict'>
>>> what_am_i = set()
>>> type(what_am_i)
<class 'set'>
>>> what_am_i
set()
```

Instead, use the `set` type converter function without an argument.

Sets contain a unique collection of elements of any type. You can add to a set using its add method, and test for membership with the `in` operator.

```
>>> set_of_numbers = {1, 2, 3, 4}
>>> set_of_numbers
{1, 2, 3, 4}
>>> set_of_numbers.add(5)
>>> set_of_numbers
{1, 2, 3, 4, 5}
>>> 3 in set_of_numbers
True
>>> 6 in set_of_numbers
False
```

Since sets hold unique collections of elements, you can use the `set` type conversion function to remove duplicates from a list.

```
>>> list_of_numbers = [1, 2, 1, 3, 4, 8, 11, 4, 5, 8]
>>> set(list_of_numbers)
{1, 2, 3, 4, 5, 8, 11}
```

6.5. Formatting Strings

This book is aimed at aspiring web developers, so much of our focus will be on creating dynamic web pages using Python. Web pages are stored in text files, which are essentially files containing a string of text. The ability to process and format text is quite important to us. That's what this section is about.

6.6. The `format` method for strings

The easiest and most powerful way to format a string in Python 3 is to use the `format` method.

```
>>> "His name is {0}!".format("Arthur")
'His name is Arthur!'
>>> name = "Alice"
>>> age = 10
>>> "I am {0} and I am {1} years old.".format(name, age)
'I am Alice and I am 10 years old.'
>>> n1 = 4
>>> n2 = 5
>>> "2 ** 10 = {0} and {1} * {2} = {3:f}".format(2 ** 10, n1, n2, n1 * n2)
'2 ** 10 = 1024 and 4 * 5 = 20.000000'
```

The key idea is that one provides a *formatter string* which contains *placeholder fields*, `... {0} ... {1} ... {2} ...` etc. The **format method** of a string uses the numbers as indexes into its arguments, and substitutes the appropriate argument into each placeholder field.

Each of the placeholders can also contain a **format specification** — it is always introduced by the `:` symbol. This can control things like

- whether the field is aligned left `<`, centered `^`, or right `>`
- the width allocated to the field within the result string (a number like `10`)
- the type of conversion (we'll initially only force conversion to float, `f`, as we did in line 11 of the code above, or perhaps we'll ask integer numbers to be converted to hexadecimal using `x`)
- if the type conversion is a float, you can also specify how many decimal places are wanted (typically, `.2f` is useful for working with currencies to two decimal places.)

You can have multiple placeholders indexing the same argument, or perhaps even have extra arguments that are not referenced at all:

```
letter = """
Dear {0} {2},

{0}, I have an interesting money-making proposition for you!
If you deposit $10 million into my bank account, I can
double your money ...
"""

print(letter.format("Paris", "Whitney", "Hilton"))
print(letter.format("Bill", "Henry", "Gates"))
```

This produces the following:

```
Dear Paris Hilton,

Paris, I have an interesting money-making proposition for you!
If you deposit $10 million into my bank account, I can
double your money ...

Dear Bill Gates,

Bill, I have an interesting money-making proposition for you!
If you deposit $10 million into my bank account I can,
double your money ...
```

As you might expect, you'll get an index error if your placeholders refer to arguments that you do not provide:

```
>>> "hello {3}".format("Dave")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: tuple index out of range
```

In addition to positional arguments in format strings, *named arguments* are also supported:

```
>>> s = "'{cat}'ll {verb1} me very much {time}, I should {verb2}!'"
>>> s.format(verb1="miss", cat="Dinah", time="to-night", verb2="think")
"'Dinah'll miss me very much to-night, I should think!'"
```

Notice that the order of the arguments to the `format` method example doesn't correspond with the order they appear in the format string. These are keywords, as with dictionaries, so their order is not relevant.

Old style format strings

Earlier versions of Python used a cryptic way to format strings. It is considered deprecated and will eventually disappear from the language.

While we won't use it in this book, you will still see it around in lots of existing Python code, so it is good to be aware of it.

The syntax for the old string formatting operation looks like this:

```
"<FORMAT>" % (<VALUES>)
```

To see how this works, here are a few examples:

```
>>> "His name is %s."  % "Arthur"
'His name is Arthur.'
>>> name = "Alice"
>>> age = 10
>>> "I am %s and I am %d years old." % (name, age)
'I am Alice and I am 10 years old.'
>>> n1 = 4
>>> n2 = 5
>>> "2**10 = %d and %d * %d = %f" % (2**10, n1, n2, n1 * n2)
'2 ** 10 = 1024 and 4 * 5 = 20.000000'
>>>
```

6.7. Files

6.7.1. About files

While a program is running, its data is stored in random access memory (RAM). RAM is extremely fast, but it is also **volatile**, which means that when the program ends, or the computer shuts down, data in RAM disappears. To make data available the next time you turn on your computer and start your program, you have to write it to a **non-volatile** storage medium, such a hard drive, usb drive, or CD-RW.

Data on non-volatile storage media is stored in named locations on the media called files. By reading and writing files, programs can save information between program runs. A file is a block of data stored in the file system of the computer's operating system.

To use a file, you have to open it. When you're done, you have to close it. When you open the file, you have to decide ahead of time whether you want to read data from the file or write data to it. If you plan to write data to the file you have to choose between starting a new version of the file or writing data at the end of what was already there. This second option for writing to the file is called **appending**. The first option destroys any previously existing data in the file.

6.7.2. The `open` function

The `open` function takes two arguments. The first is the name of the file, and the second is the **mode**. Mode `'w'` means that we are opening the file for writing. Mode `'r'` means reading, and mode `'a'` means appending.

Let's begin with an example that shows these three modes in operation:

```
>>> myfile = open('test.txt', 'w')
>>> myfile.write('My first file written from Python\n')
34
>>> myfile.write('---------------------------------\n')
34
>>> myfile.write('Hello, world!')
13
>>> myfile.close()
>>> myfile = open('test.txt', 'r')
>>> contents = myfile.read()
>>> myfile.close()
>>> print(contents)
My first file written from Python
---------------------------------
Hello, world!
```

Opening a file creates what we call a file descriptor. In this example, the variable `myfile` refers to the new descriptor object. Our program calls methods on the descriptor, and this makes changes to the actual file which is located in non-volatile storage.

The first line opens the `test.txt` for writing. If there is no file named `test.txt` on the disk, it will be created. If there already is one, it will be replaced by the file we are writing and any previous data in it will be lost.

To put data in the file we invoke the `write` method on the file descriptor. We do this three times in the example above, but in bigger programs, the three separate calls to `write` will usually be replaced by a loop that writes many more lines into the file. The `write` method returns the number of bytes (characters) written to the file.

Closing the file handle tells the system that we are done writing and makes the disk file available for reading by other programs (or by our own program).

We finish this example by openning `test.txt` for reading. We then call the `read` method, assigning the contents of the file, which is a string, to a variable named `contents`, and finally print `contents` to see that it is indeed what we wrote to the file previously.

If we want to add to an already existing file, use the *append* mode.

```
>>> myfile = open('test.txt', 'a')
>>> myfile.write('\nOoops, I forgot to add this line ;-)')
37
>>> myfile.close()
>>> myfile = open('test.txt', 'r')
>>> print(myfile.read())
My first file written from Python
---------------------------------
Hello, world!
Ooops, I forgot to add this line ;-)
>>>
```

6.8. Opening a file that doesn't exist

If we try to open a file that doesn't exist, we get an error:

```
>>> f = open('wharrah.txt', 'r')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: 'wharrah.txt'
>>>
```

There is nothing wrong with the syntax of the line that resulted in the error. The error occurred because the file did not exist. Errors like these are called **exceptions**. Most modern programming languages provide support for dealing with situations like this. The process is called exception handling.

In Python, exceptions are handled with the `try ... except` statement.

```
try:
    f = open('thefile.txt', 'r')
    mydata = f.read()
    f.close()
except IOError:
    mydata = ''
```

In this example we *try* to open the data file for reading. If it succeeds, we use the `read()` method to read the file contents as a string into the variable `mydata` and close the file. If an `IOError` exception occurs, we still create `mydata` as an empty string and continue on with the program.

6.9. Reading data from files

Python file descriptors have three methods for reading in data from a file. We've already seen the `read()` method, which returns the entire contents of the file as a single string. For really big files this may not be what you want.

The `readline()` method returns one line of the file at a time. Each time you call it `readline()` returns the next line. Calls made to `readline()` after reaching the end of the file return an empty string (`''`).

```
f = open('test.txt', 'r')
while True:                             # keep reading forever
    theline = f.readline()              # try to read next line
    if len(theline) == 0:               # if there are no more lines
        break                           # leave the loop

    # Now process the line we've just read
    print(theline, end='')

f.close()
```

This is a handy pattern for our toolbox. In bigger programs, we'd squeeze more extensive logic into the body of the loop at line 8 — for example, if each line of the file contained the name and email address of one of our friends, perhaps we'd split the line into some pieces and call a function to send the friend a party invitation.

On line 8 we suppress the newline character that `print` usually appends to our strings. Why? This is because the string already has its own newline: the `readline` method in line 3 returns everything up to *and including* the newline character. This also explains the end-of-file detection logic: when there are no more lines to be read from the file, `readline` returns an empty string — one that does not even have a newline at the end, hence it's length is 0.

6.9.1. Turning a file into a list of lines

It is often useful to fetch data from a disk file and turn it into a list of lines. Suppose we have a file containing our friends and their email addresses, one per line in the file. But we'd like the lines sorted into alphabetical order. A good plan is to read everything into a list of lines, then sort the list, and then write the sorted list back to another file:

```
f1 = open('friends.txt', 'r')
friends_list = f.readlines()
f1.close()

friends_list.sort()

f2 = open('sortedfriends.txt', 'w')
for friend in friends_list:
    f2.write(v)
f2.close()
```

The `readlines` method in line 2 reads all the lines and returns a list of the strings.

We could have used the template from the previous section to read each line one-at-a-time, and to build up the list ourselves, but it is a lot easier to use the method that the Python implementors gave us!

6.9.2. An example

Many useful line-processing programs will read a text file line-at-a-time and do some minor processing as they write the lines to an output file. They might number the lines in the output file, or insert extra blank lines after every 60 lines to make it convenient for

printing on sheets of paper, or extract some specific columns only from each line in the source file, or only print lines that contain a specific substring. We call this kind of program a **filter**.

Here is a filter that copies one file to another, omitting any lines that begin with `#` :

```
infile = open(oldfile, 'r')
outfile = open(newfile, 'w')
while True:
    text = infile.readline()
    if len(text) == 0:
        break
    if text[0] == '#':
        continue

    # put any more processing logic here
    outfile.write(text)

infile.close()
outfile.close()
```

The `continue` statement at line 9 skips over the remaining lines in the current iteration of the loop, but the loop will still iterate. This style looks a bit contrived here, but it is often useful to say *"get the lines we're not concerned with out of the way early, so that we have cleaner more focussed logic in the meaty part of the loop that might be written around line 11."*

Thus, if `text` is the empty string, the loop exits. If the first character of `text` is a hash mark, the flow of execution goes to the top of the loop, ready to start processing the next line. Only if both conditions fail do we fall through to do the processing at line 11, in this example, writing the line into the new file.

Let's consider one more case: suppose your original file contained empty lines. At line 6 above, would this program find the first empty line in the file, and terminate immediately? No! Recall that `readline` always includes the newline character in the string it returns. It is only when we try to read *beyond* the end of the file that we get back the empty string of length 0.

6.10. `repr()` and `eval()` functions

Python has a built-in function named `repr` that takes a Python object as an argument and returns a string *representation* of that object. For Python's built-in types, the string representation of an object can be *evaluated* using the built-in `eval` function to recreate the object.

The way this works is easiest to demonstrate by example.

```
>>> mylist = [1, 2, 'buckle', 'my', 'shoe']
>>> type(mylist)
<class 'list'>
>>> repr(mylist)
"[1, 2, 'buckle', 'my', 'shoe']"
>>> s = repr(mylist)
>>> s
"[1, 2, 'buckle', 'my', 'shoe']"
>>> type(s)
<class 'str'>
>>> eval(s)
[1, 2, 'buckle', 'my', 'shoe']
>>> thing = eval(s)
>>> thing
[1, 2, 'buckle', 'my', 'shoe']
>>> type(thing)
<class 'list'>
>>>
```

The list object, `mylist` is converted into a *string representation* using the `repr` function, and this string representation is than converted back into a Python list object using the `eval` function (which *evaluates* the string representation).

While we will learn much better ways to achieve the goal of storing Python objects into data files later, `repr` and `eval` provide us with an easy to understand tool for writing and then reading back Python data to files that we can use now.

6.11. Modules

A **module** is a file containing Python definitions and statements intended for use in other Python programs. There are many Python modules that come with Python as part of the **standard library**. We have seen two of these already, the `doctest` module and the `string` module.

6.12. Creating modules

All we need to create a module is a text file with a `.py` extension on the filename:

```
#  seqtools.py
#
def remove_at(pos, seq):
    return seq[:pos] + seq[pos+1:]
```

We can now use our module in both scripts and the Python shell. To do so, we must first *import* the module. There are two ways to do this:

```
>>> from seqtools import remove_at
>>> s = "A string!"
>>> remove_at(4, s)
'A sting!'
```

and:

```
>>> import seqtools
>>> s = "A string!"
>>> seqtools.remove_at(4, s)
'A sting!'
```

In the first example, `remove_at` is called just like the functions we have seen previously. In the second example the name of the module and a dot (.) are written before the function name.

Notice that in either case we do not include the `.py` file extension when importing. Python expects the file names of Python modules to end in `.py`, so the file extention is not included in the **import statement**.

The use of modules makes it possible to break up very large programs into manageable sized parts, and to keep related parts together.

6.13. Namespaces

A **namespace** is a syntactic container which permits the same name to be used in different modules or functions (and as we will see soon, in classes and methods).

Each module determines its own namespace, so we can use the same name in multiple modules without causing an identification problem.

```
# module1.py

question = "What is the meaning of life, the Universe, and everything?"
answer = 42
```

```
# module2.py

question = "What is your quest?"
answer = "To seek the holy grail."
```

We can now import both modules and access `question` and `answer` in each:

```
>>> import module1
>>> import module2
>>> print module1.question
What is the meaning of life, the Universe, and everything?
>>> print module2.question
What is your quest?
>>> print module1.answer
42
>>> print module2.answer
To seek the holy grail.
>>>
```

If we had used `from module1 import *` and `from module2 import *` instead, we would have a **naming collision** and would not be able to access `question` and `answer` from `module1`.

Functions also have their own namespace:

```
def f():
    n = 7
    print("printing n inside of f: {0}".format(n))

def g():
    n = 42
    print("printing n inside of g: {0}".format(n))

n = 11
print("printing n before calling f: {0}".format(n))
f()
print("printing n after calling f: {0}".format(n))
g()
print("printing n after calling g: {0}".format(n))
```

Running this program produces the following output:

```
printing n before calling f: 11
printing n inside of f: 7
printing n after calling f: 11
printing n inside of g: 42
printing n after calling g: 11
```

The three `n` 's here do not collide since they are each in a different namespace.

Namespaces permit several programmers to work on the same project without having naming collisions.

6.14. Attributes and the dot operator

Variables defined inside a module are called **attributes** of the module. They are accessed by using the **dot operator** (`.`). The `question` attribute of `module1` and `module2` are accessed using `module1.question` and `module2.question` .

Modules contain functions as well as attributes, and the dot operator is used to access them in the same way. `seqtools.remove_at` refers to the `remove_at` function in the `seqtools` module.

In Chapter 7 we introduced the `find` function from the `string` module. The `string` module contains many other useful functions:

```
>>> import string
>>> string.capitalize('maryland')
'Maryland'
>>> string.capwords("what's all this, then, amen?")
"What's All This, Then, Amen?"
>>> string.center('How to Center Text Using Python', 70)
'                   How to Center Text Using Python                    '
>>> string.upper('angola')
'ANGOLA'
>>>
```

You should use pydoc to browse the other functions and attributes in the string module.

6.15. Glossary

dictionary
A collection of key-value pairs that maps from keys to values. The keys can be any immutable type, and the values can be any type.

file
A named entity, usually stored on a hard drive, floppy disk, or CD-ROM, that contains a stream of characters.

file system
A method for naming, accessing, and organizing files and the data they contain.

fully qualified name
A name that is prefixed by some namespace identifier and the dot operator, or by an instance object, e.g. `math.sqrt` or `f.open('myfile.txt', 'r')` .

handle
An object in our program that is connected to an underlying resource (e.g. a file). The file handle lets our program manipulate / read/ write / close the actual file that is on our disk.

import
A statement which permits functions and variables defined in a Python script to be brought into the environment of another script or a running Python shell.For example, assume the following is in a script named `tryme.py` :

```
def print_thrice(thing):
    print thing, thing, thing

n = 42
s = "And now for something completely different..."
```

Now begin a python shell from within the same directory where `tryme.py` is located:

```
$ ls
tryme.py
$ python
>>>
```

Three names are defined in `tryme.py` : `print_thrice` , `n` , and `s` . If we try to access any of these in the shell without first importing, we get an error:

```
>>> n
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
>>> print_thrice("ouch!")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'print_thrice' is not defined
```

If we import everything from `tryme.py` , however, we can use everything defined in it:

```
>>> from tryme import *
>>> n
42
>>> s
'And now for something completely different...'
>>> print_thrice("Yipee!")
Yipee! Yipee! Yipee!
>>>
```

Note that you do not include the `.py` from the script name in the import statement.

key
A data item that is *mapped to* a value in a dictionary. Keys are used to look up values in a dictionary.

key-value pair
One of the pairs of items in a dictionary. Values are looked up in a dictionary by key.

mapping type
A mapping type is a data type comprised of a collection of keys and associated values. Python's only built-in mapping type is the dictionary. Dictionaries implement the associative array abstract data type.

mode
A distinct method of operation within a computer program. Files in Python can be openned in one of three modes: read (`'r'`), write (`'w'`), and append (`'a'`).

non-volatile memory
Memory that can maintain its state without power. Hard drives, flash drives, and rewritable compact disks (CD-RW) are each examples of non-volatile memory.

path
A sequence of directory names that specifies the exact location of a file.

set
A collection of unique, unordered elements.

text file
A file that contains printable characters organized into lines separated by newline characters.

volatile memory
Memory which requires an electrical current to maintain state. The *main memory* or RAM of a computer is volatile. Information stored in RAM is lost when the computer is turned off.