정보교육을 위한 파이썬

정보 탐색

Version 0.0.9-d2

저자: Charles Severance
번역: 이광춘, 한정수
(xwmooc)

Printing history:

October 2013: Major revision to Chapters 13 and 14 to switch to JSON and use OAuth. Added new chapter on Visualization.

September 2013: Published book on Amazon CreateSpace

January 2010: Published book using the University of Michigan Espresso Book machine.

December 2009: Major revision to chapters 2-10 from *Think Python: How to Think Like a Computer Scientist* and writing chapters 1 and 11-15 to produce *Python for Informatics: Exploring Information*

June 2008: Major revision, changed title to *Think Python: How to Think Like a Computer Scientist*.

August 2007: Major revision, changed title to *How to Think Like a (Python) Programmer*.

April 2002: First edition of *How to Think Like a Computer Scientist*.

The LaTeX source for the *Think Python: How to Think Like a Computer Scientist* version of this book is available from `http://www.thinkpython.com`.

Chapter 1

리스트

1.1 리스트는 열이다.

문자열처럼, **리스트(list)**는 일련의 값이다. 문자열에서, 값은 문자지만, 리스트에서는 임의의 형(type)이 될 수 있다. 리스트의 값은 **요소(elements)**나 때때로 **항목(items)**으로 불린다.

신규 리스틀 생성하는 방법은 여러가지다. 가장 간단한 방법은 꺾쇠 괄호(`[` 와 `]`)로 요소를 감싸는 것이다.

```
[10, 20, 30, 40]
['crunchy frog', 'ram bladder', 'lark vomit']
```

첫번째 예제는 네 개의 정수 리스트다. 드번째 예제는 3개의 문자열 리스트다. 문자열의 요소는 같은 형(type)일 필요는 없다. 다음의 리스트는 문자열, 부동소수점 숫자, 정수, (아!) 또 다른 리스트를 담고 있다.

```
['spam', 2.0, 5, [10, 20]]
```

또 다른 리스트 내부에 리스트는 **중첩(nested)**되어 있다.

어떤 요소도 담고 있지 않는 리스트는 빈 리스트(empty list)라고 부르고, 빈 꺾쇠 괄호("[]")로 생성할 수 있다.

예상했듯이, 리스트 값을 변수에 할당할 수 있다.

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> numbers = [17, 123]
>>> empty = []
>>> print cheeses, numbers, empty
['Cheddar', 'Edam', 'Gouda'] [17, 123] []
```

1.2 리스트는 변경가능하다.

리스트의 요소에 접근하는 구문은 문자열의 문자에 접근하는 것과 동일한 꺾쇠 괄호 연산자다. 꺽쇠 괄호 내부의 표현식은 인덱스를 명세한다. 인덱스는 0에서부터 시작한다.

```
>>> print cheeses[0]
Cheddar
```

문자열과 달리, 리스트의 항목의 순서를 바꾸거나, 리스트에 새로운 항목을 재할당할 수 있기 때문에 리스트는 변경가능하다. 꺾쇠 괄호 연산자가 할당문의 왼쪽편에 나타날 때, 새로 할당될 리스트의 요소를 나타낸다.

```
>>> numbers = [17, 123]
>>> numbers[1] = 5
>>> print numbers
[17, 5]
```

리스트 numbers의 1번째 요소는 123 값을 가지고 있으나, 이제는 5 값을 가진다.

리스트를 인텍스와 요소의 관계로 생각할 수 있다. 이 관계를 **매핑(mapping)**이라고 부른다. 각각의 인덱스는 요소중의 하나에 대응("maps to")된다.

리스트 인덱스는 문자열 인덱스와 같은 방식으로 동작한다.

- 임의의 정수 표현식은 인덱스로 사용될 수 있다.
- 존재하지 않는 요소를 읽거나 쓰려고 하면, IndexError가 발생한다.
- 인덱스가 음의 값이면, 리스트의 끝에서부터 역으로 센다.

in 연산자도 또한 리스트에서 동작한다.

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses
True
>>> 'Brie' in cheeses
False
```

1.3 리스트 운행법

리스트의 요소를 운행하는 가장 흔한 방법은 for문을 사용하는 것이다. 구문은 문자열에서 사용한 것과 동일하다.

```
for cheese in cheeses:
    print cheese
```

리스트의 요소를 읽기만 한다면 이것만으로 잘 동작한다. 하지만, 리스트의 요소를 쓰거나, 갱신하는 경우, 인텍스가 필요하다. 리스트의 요소를 쓰거나 갱신하는 흔한 방법은 range와 len 함수를 조합하는 것이다.

```
for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2
```

상기 루프는 리스트를 운행하고 각 요소를 갱신한다. len함수는 리스트의 요소의 갯수를 반환한다. range 함수는 0 에서 $n-1$ 까지 리스트 인텍스를 반환한다.

여기서, *n*은 리스트의 길이다. 매번 루프가 반복될 때마다, i는 다음 요소의 인덱스를 얻는다. 몸통 부문의 할당문은 i를 사용해서 요소의 옛값을 일고 새값을 할당한다.

빈 리스트의 for문은 결코 몸통부분을 실행하지 않는다.

```
for x in empty:
    print 'This never happens.'
```

리스트가 또 다른 리스트를 담을 수 있지만, 중첩된 리스트는 여전히 하나의 요소로 센다. 다음 리스트의 길이는 4이다.

```
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

1.4 리스트 연산자

+ 연산자는 리스트를 결합한다.

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print c
[1, 2, 3, 4, 5, 6]
```

유사하게 * 연산자는 주어진 횟수 만큼 리스트를 반복한다.

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

첫 예제는 [0]을 4회 반복한다. 두 번째 예제는 [1, 2, 3] 리스트를 3회 반복한다.

1.5 리스트 쪼개기(List slices)

쪼개는 연산자는 리스트에도 동작한다.

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

첫 번째 인덱스를 생략하면, 쪼개기는 처음부터 시작한다. 두 번째 인덱스를 생략하면, 쪼개기는 끝까지 간다. 그래서 양쪽의 인덱스를 생략하면, 쪼개기는 전체 리스트를 복사한다.

```
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

리스트는 변경이 가능하기 때문에 리스트를 접고, 돌리고, 훼손하는 연ㅅ나을 수행하기 전에 사본을 만드는 것이 유용하다.

할당문의 왼편의 쪼개기 연산자는 복수의 요소를 갱신할 수 있다.

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']
>>> print t
['a', 'x', 'y', 'd', 'e', 'f']
```

1.6 리스트 메쏘드

파이썬은 리스트에 연산하는 메쏘드를 제공한다. 예를 들어, `append` 메쏘드는 리스트 끝에 신규 요소를 추가한다.

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> print t
['a', 'b', 'c', 'd']
```

`extend` 메쏘드는 인수로 리스트를 받아 모든 요소를 리스트에 추가한다.

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> print t1
['a', 'b', 'c', 'd', 'e']
```

상기 예제는 `t2` 리스트를 변경없이 놓아둔다.

`sort` 메쏘드는 낮음에서 높음으로 리스트의 요소를 정렬한다.

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> print t
['a', 'b', 'c', 'd', 'e']
```

대부분의 리스트 메쏘드는 보이드(void)여서, 리스트를 변경하고 `None`을 반환한다. 우연히 `t = t.sort()` 이렇게 작성한다면, 결과에 실망할 것이다.

1.7 요소 삭제

리스트에서 요소를 삭제하는 몇 가지 방법이 있다. 리스트 요소 인덱스를 알고 있다면, `pop` 메쏘드를 사용한다.

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> print t
['a', 'c']
>>> print x
b
```

pop 메쏘드는 리스트를 변경하여 제거된 요소를 반환한다. 인덱스를 주지 않으면, 마지막 요소를 지우고 반환한다.

요소에서 제거된 값이 필요없다면, del 연산자를 사용할 수 있다.

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> print t
['a', 'c']
```

인덱스가 아닌 제거할 요소를 알고 있다면, remove 메쏘드를 사용할 수 있다.

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> print t
['a', 'c']
```

remove 메쏘드의 반환값은 None이다.

하나 이상의 요소를 제거하기 위해서, 쪼개기 인덱스(slice index)와 del을 사용한다.

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
>>> print t
['a', 'f']
```

마찬가지로, 쪼개기는 두 번째 인덱스를 포함하지 않는 두 번째 인덱스까지의 모든 요소를 선택한다.

1.8 리스트와 함수

루프를 작성하지 않고 리스트를 빠를게 살펴볼 수 있도록 리스트에 적용할 수 있는 많은 내장함수가 있다.

```
>>> nums = [3, 41, 12, 9, 74, 15]
>>> print len(nums)
6
>>> print max(nums)
74
>>> print min(nums)
3
>>> print sum(nums)
154
>>> print sum(nums)/len(nums)
25
```

리스트 요소가 숫자일 때, sum() 함수는 동작한다. max(), len(), 등등의 함수는 문자열 리스트나, 비교가능한 다른 형(type)의 리스트에 사용할 수 있다.

리스트를 사용해서, 사용자가 입력한 숫자 목록의 평균을 계산하는 앞서 작성한 프로그램을 다시 작성할 수 있다.

우선 리스트 없이 평균을 계산하는 프로그램:

```
total = 0
count = 0
while ( True ) :
    inp = raw_input('Enter a number: ')
    if inp == 'done' : break
    value = float(inp)
    total = total + value
    count = count + 1

average = total / count
print 'Average:', average
```

이 프로그램에서, count 와 sum 변수를 사용해서 반복적으로 사용자가 숫자를 입력하면 값을 저장하고, 지금까지 사용자가 입력한 누적 합계를 계산하는 것이다.

단순하게, 사용자가 입력한 각 숫자를 기억하고 내장함수를 사용해서 프로그램 마지막에 합계와 갯수를 계산한다.

```
numlist = list()
while ( True ) :
    inp = raw_input('Enter a number: ')
    if inp == 'done' : break
    value = float(inp)
    numlist.append(value)

average = sum(numlist) / len(numlist)
print 'Average:', average
```

루프가 시작되기 전에 빈 리스트를 생성하고, 매번 숫자를 입력할 때, 숫자를 리스트에 추가한다. 프로그램 마지막에 간단하게 리스트의 합계를 계산하고, 평균을 출력하기 위해서 입력한 숫자 개수로 나누었다.

1.9 리스트와 문자열

문자열은 일련의 문자이고, 리스트는 일련의 값이다. 하지만 리스트의 문자는 문자열과 같지는 않다. 문자열에서 리스트의 문자로 변환하기 위해서, list를 사용한다.

```
>>> s = 'spam'
>>> t = list(s)
>>> print t
['s', 'p', 'a', 'm']
```

list는 내장함수의 이름이기 때문에, 변수명으로 사용하는 것을 피해야 한다. l의 사용을 1 처럼 보이기 때문에 피한다. 그래서, t를 사용하였다.

list 함수는 문자열을 각각의 문자로 쪼갠다. 문자열을 단어로 쪼개려면, split 메쏘드를 사용할 수 있다.

```
>>> s = 'pining for the fjords'
>>> t = s.split()
>>> print t
['pining', 'for', 'the', 'fjords']
>>> print t[2]
the
```

split 메쏘드를 사용해서 문자열을 리스트의 토큰으로 쪼개기만 하면, 인덱스 연산자('[]')를 사용하여 리스트의 특정한 단어를 볼 수 있다.

구분자(delimiter)가 단어의 경계로 어느 문자를 사용할지를 지정하는데, split 메쏘드를 호출할 때 두 번째 선택 인수로 사용할 수 있다. 다음 예제는 구분자로 하이픈('-')을 사용한다.

```
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
>>> s.split(delimiter)
['spam', 'spam', 'spam']
```

join 메쏘드는 split 메쏘드의 역이다. 문자열 리스트를 받아 리스트 요소를 결합한다. join은 문자열 메쏘드여서, 구분자를 호출하여 매개 변수로 넘길 수 있다.

```
>>> t = ['pining', 'for', 'the', 'fjords']
>>> delimiter = ' '
>>> delimiter.join(t)
'pining for the fjords'
```

상기의 경우, 구분자가 공백 문자여서 join 메쏘드는 단어 사이에 공백을 넣는다. 공백없이 문자열을 결합하기 위해서, 구분자로 빈 문자열 ''을 사용한다.

1.10 Parsing lines

Usually when we are reading a file we want to do something to the lines other than just printing the whole line. Often we want to find the "interesting lines" and then **parse** the line to find some interesting *part* of the line. What if we wanted to print out the day of the week from those lines that start with "From ".

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
```

The split method is very effective when faced with this kind of problem. We can write a small program that looks for lines where the line starts with "From " and then split those lines and then print out the third word in the line:

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if not line.startswith('From ') : continue
    words = line.split()
    print words[2]
```

We also use the contracted form of the `if` statement where we put the `continue` on the same line as the `if`. This contracted form of the `if` functions the same as if the `continue` were on the next line and indented.

The program produces the following output:

```
Sat
Fri
Fri
Fri
    ...
```

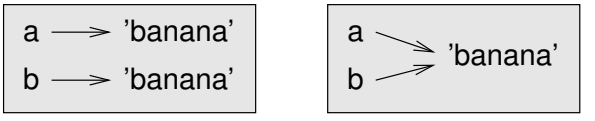
Later, we will learn increasingly sophisticated techniques for picking the lines to work on and how we pull those lines apart to find the exact bit of information we are looking for.

1.11 Objects and values

If we execute these assignment statements:

```
a = 'banana'
b = 'banana'
```

We know that `a` and `b` both refer to a string, but we don't know whether they refer to the *same* string. There are two possible states:



In one case, `a` and `b` refer to two different objects that have the same value. In the second case, they refer to the same object.

To check whether two variables refer to the same object, you can use the `is` operator.

```
>>> a = 'banana'
>>> b = 'banana'
>>> a is b
True
```

In this example, Python only created one string object, and both `a` and `b` refer to it.

But when you create two lists, you get two objects:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
```

In this case we would say that the two lists are **equivalent**, because they have the same elements, but not **identical**, because they are not the same object. If two objects are identical, they are also equivalent, but if they are equivalent, they are not necessarily identical.

Until now, we have been using "object" and "value" interchangeably, but it is more precise to say that an object has a value. If you execute `a = [1,2,3]`, `a` refers to a list object whose value is a particular sequence of elements. If another list has the same elements, we would say it has the same value.

1.12 Aliasing

If `a` refers to an object and you assign `b = a`, then both variables refer to the same object:

```
>>> a = [1, 2, 3]
>>> b = a
>>> b is a
True
```

The association of a variable with an object is called a **reference**. In this example, there are two references to the same object.

An object with more than one reference has more than one name, so we say that the object is **aliased**.

If the aliased object is mutable, changes made with one alias affect the other:

```
>>> b[0] = 17
>>> print a
[17, 2, 3]
```

Although this behavior can be useful, it is error-prone. In general, it is safer to avoid aliasing when you are working with mutable objects.

For immutable objects like strings, aliasing is not as much of a problem. In this example:

```
a = 'banana'
b = 'banana'
```

It almost never makes a difference whether `a` and `b` refer to the same string or not.

1.13 List arguments

When you pass a list to a function, the function gets a reference to the list. If the function modifies a list parameter, the caller sees the change. For example, `delete_head` removes the first element from a list:

```
def delete_head(t):
    del t[0]
```

Here's how it is used:

```
>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> print letters
['b', 'c']
```

The parameter `t` and the variable `letters` are aliases for the same object.

It is important to distinguish between operations that modify lists and operations that create new lists. For example, the `append` method modifies a list, but the `+` operator creates a new list:

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> print t1
[1, 2, 3]
>>> print t2
None

>>> t3 = t1 + [3]
>>> print t3
[1, 2, 3]
>>> t2 is t3
False
```

This difference is important when you write functions that are supposed to modify lists. For example, this function *does not* delete the head of a list:

```
def bad_delete_head(t):
    t = t[1:]               # WRONG!
```

The slice operator creates a new list and the assignment makes `t` refer to it, but none of that has any effect on the list that was passed as an argument.

An alternative is to write a function that creates and returns a new list. For example, `tail` returns all but the first element of a list:

```
def tail(t):
    return t[1:]
```

This function leaves the original list unmodified. Here's how it is used:

```
>>> letters = ['a', 'b', 'c']
>>> rest = tail(letters)
>>> print rest
['b', 'c']
```

Exercise 1.1 Write a function called `chop` that takes a list and modifies it, removing the first and last elements, and returns `None`.

Then write a function called `middle` that takes a list and returns a new list that contains all but the first and last elements.

1.14 Debugging

Careless use of lists (and other mutable objects) can lead to long hours of debugging. Here are some common pitfalls and ways to avoid them:

1. Don't forget that most list methods modify the argument and return `None`. This is the opposite of the string methods, which return a new string and leave the original alone.

   If you are used to writing string code like this:

   ```
   word = word.strip()
   ```

   It is tempting to write list code like this:

   ```
   t = t.sort()           # WRONG!
   ```

   Because `sort` returns `None`, the next operation you perform with `t` is likely to fail.

   Before using list methods and operators, you should read the documentation carefully and then test them in interactive mode. The methods and operators that lists share with other sequences (like strings) are documented at `docs.python.org/lib/typesseq.html`. The methods and operators that only apply to mutable sequences are documented at `docs.python.org/lib/typesseq-mutable.html`.

2. Pick an idiom and stick with it.

   Part of the problem with lists is that there are too many ways to do things. For example, to remove an element from a list, you can use `pop`, `remove`, `del`, or even a slice assignment.

   To add an element, you can use the `append` method or the `+` operator. But don't forget that these are right:

   ```
   t.append(x)
   t = t + [x]
   ```

   And these are wrong:

   ```
   t.append([x])          # WRONG!
   t = t.append(x)        # WRONG!
   t + [x]                # WRONG!
   t = t + x              # WRONG!
   ```

Try out each of these examples in interactive mode to make sure you understand what they do. Notice that only the last one causes a runtime error; the other three are legal, but they do the wrong thing.

3. Make copies to avoid aliasing.

   If you want to use a method like `sort` that modifies the argument, but you need to keep the original list as well, you can make a copy.

   ```
   orig = t[:]
   t.sort()
   ```

   In this example you could also use the built-in function `sorted`, which returns a new, sorted list and leaves the original alone. But in that case you should avoid using `sorted` as a variable name!

4. Lists, `split`, and files

   When we read and parse files, there are many opportunities to encounter input that can crash our program so it is a good idea to revisit the **guardian** pattern when it comes writing programs that read through a file and look for a "needle in the haystack".

   Let's revisit our program that is looking for the day of the week on the from lines of our file:

   ```
   From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
   ```

   Since we are breaking this line into words, we could dispense with the use of `startswith` and simply look at the first word of the line to determine if we are interested in the line at all. We can use `continue` to skip lines that don't have "From" as the first word as follows:

   ```
   fhand = open('mbox-short.txt')
   for line in fhand:
       words = line.split()
       if words[0] != 'From' : continue
       print words[2]
   ```

   This looks much simpler and we don't even need to do the `rstrip` to remove the newline at the end of the file. But is it better?

   ```
   python search8.py
   Sat
   Traceback (most recent call last):
     File "search8.py", line 5, in <module>
       if words[0] != 'From' : continue
   IndexError: list index out of range
   ```

   It kind of works and we see the day from the first line (Sat) but then the program fails with a traceback error. What went wrong? What messed-up data caused our elegant, clever and very Pythonic program to fail?

You could stare at it for a long time and puzzle through it or ask someone for help, but the quicker and smarter approach is to add a `print` statement. The best place to add the print statement is right before the line where the program failed and print out the data that seems to be causing the failure.

Now this approach may generate a lot of lines of output but at least you will immediately have some clue as to the problem at hand. So we add a print of the variable `words` right before line five. We even add a prefix "Debug:" to the line so we can keep our regular output separate from our debug output.

```
for line in fhand:
    words = line.split()
    print 'Debug:', words
    if words[0] != 'From' : continue
    print words[2]
```

When we run the program, a lot of output scrolls off the screen but at the end, we see our debug output and the traceback so we know what happened just before the traceback.

```
Debug: ['X-DSPAM-Confidence:', '0.8475']
Debug: ['X-DSPAM-Probability:', '0.0000']
Debug: []
Traceback (most recent call last):
  File "search9.py", line 6, in <module>
    if words[0] != 'From' : continue
IndexError: list index out of range
```

Each debug line is printing the list of words which we get when we `split` the line into words. When the program fails the list of words is empty `[]`. If we open the file in a text editor and look at the file, at that point it looks as follows:

```
X-DSPAM-Result: Innocent
X-DSPAM-Processed: Sat Jan  5 09:14:16 2008
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000

Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
```

The error occurs when our program encounters a blank line! Of course there are "zero words" on a blank line. Why didn't we think of that when we were writing the code. When the code looks for the first word (`word[0]`) to check to see if it matches "From", we get an "index out of range" error.

This of course is the perfect place to add some **guardian** code to avoid checking the first word if the first word is not there. There are many ways to protect this code, we will choose to check the number of words we have before we look at the first word:

```
fhand = open('mbox-short.txt')
count = 0
for line in fhand:
    words = line.split()
    # print 'Debug:', words
    if len(words) == 0 : continue
    if words[0] != 'From' : continue
    print words[2]
```

First we commented out the debug print statement instead of removing it in case our modification fails and we need to debug again. Then we added a guardian statement that checks to see if we have zero words, and if so, we use `continue` to skip to the next line in the file.

We can think of the two `continue` statements as helping us refine the set of lines which are "interesting" to us and which we want to process some more. A line which has no words is "uninteresting" to us so we skip to the next line. A line which does not have "From" as its first word is uninteresting to us so we skip it.

The program as modified runs successfully so perhaps it is correct. Our guardian statement does make sure that the `words[0]` will never fail, but perhaps it is not enough. When we are programming, we must always be thinking, "What might go wrong?".

Exercise 1.2 Figure out which line of the above program is still not properly guarded. See if you can construct a text file which causes the program to fail and then modify the program so that the line is properly guarded and test it to make sure it handles your new text file.

Exercise 1.3 Rewrite the guardian code in the above example without two `if` statements. Instead use a compound logical expression using the `and` logical operator with a single `if` statement.

1.15 Glossary

aliasing: A circumstance where two or more variables refer to the same object.

delimiter: A character or string used to indicate where a string should be split.

element: One of the values in a list (or other sequence), also called items.

equivalent: Having the same value.

index: An integer value that indicates an element in a list.

identical: Being the same object (which implies equivalence).

list: A sequence of values.

list traversal: The sequential accessing of each element in a list.

nested list: A list that is an element of another list.

object: Something a variable can refer to. An object has a type and a value.

reference: The association between a variable and its value.

1.16 Exercises

Exercise 1.4 Download a copy of the file from `www.py4inf.com/code/romeo.txt`

Write a program to open the file `romeo.txt` and read it line by line. For each line, split the line into a list of words using the `split` function.

For each word, check to see if the word is already in a list. If the word is not in the list, add it to the list.

When the program completes, sort and print the resulting words in alphabetical order.

```
Enter file: romeo.txt
['Arise', 'But', 'It', 'Juliet', 'Who', 'already',
'and', 'breaks', 'east', 'envious', 'fair', 'grief',
'is', 'kill', 'light', 'moon', 'pale', 'sick', 'soft',
'sun', 'the', 'through', 'what', 'window',
'with', 'yonder']
```

Exercise 1.5 Write a program to read through the mail box data and when you find line that starts with "From", you will split the line into words using the `split` function. We are interested in who sent the message which is the second word on the From line.

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
```

You will parse the From line and print out the second word for each From line and then you will also count the number of From (not From:) lines and print out a count at the end.

This is a good sample output with a few lines removed:

```
python fromcount.py
Enter a file name: mbox-short.txt
stephen.marquard@uct.ac.za
louis@media.berkeley.edu
zqian@umich.edu

[...some output removed...]

ray@media.berkeley.edu
```

```
cwen@iupui.edu
cwen@iupui.edu
cwen@iupui.edu
There were 27 lines in the file with From as the first word
```

Exercise 1.6 Rewrite the program that prompts the user for a list of numbers and prints out the maximum and minimum of the numbers at the end when the user enters "done". Write the program to store the numbers the user enters in a list and use the `max()` and `min()` functions to compute the maximum and minimum numbers after the loop completes.

```
Enter a number: 6
Enter a number: 2
Enter a number: 9
Enter a number: 3
Enter a number: 5
Enter a number: done
Maximum: 9.0
Minimum: 2.0
```