Introduction to Python

Python: very high level language, has high-level data structures built-in (such as dynamic arrays and dictionaries). easy to use and learn. Can be used for scripting (instead of Perl, awk), or for general programming. It's an interpreted language.

Trivia: The language is named after the BBC show "Monty Python's Flying Circus" and has nothing to do with reptiles. (Making references to Monty Python skits in documentation is not only allowed, it is encouraged).

This document includes a small set of the functions in Python, just enough to give a taste for the language and enable you to do the lab. Most example are cut and paste from the Python tutorial below:

See

The Python Tutorial — Python v2.7 documentation (http://docs.python.org/tutorial/)

Reference guide for built-in functions: http://docs.python.org/library/functions.html#open

Starting Python

To invoke the interpreter, type python to the shell.

The interpreter operates like the Unix shell: when called with standard input connected to a terminal, it reads and executes commands interactively; when called with a file name argument or with a file as standard input, it reads and executes a *script* from that file.

Using Python as a Calculator

First, let's play and use python as a calculator::

```
>>> 2+2
4
>>> # This is a comment
... 2+2
4
>>> 2+2  # and a comment on the same line as code
4
>>> (50-5*6)/4
5
>>> # Integer division returns the floor:
... 7/3
2
>>> 7/-3
-3
```

The equal sign (`'='`) is used to assign a value to a variable. Afterwards, no result is displayed before the next interactive prompt:

```
>>> width = 20
>>> height = 5*9
>>> width * height
900
```

A value can be assigned to several variables simultaneously:

```
>>> x = y = z = 0  # Zero x, y and z
>>> x
0
>>> y
0
>>> z
0
```

There is full support for floating point; operators with mixed type operands convert the integer operand to floating point:

```
>>> 3 * 3.75 / 1.5
7.5
>>> 7.0 / 2
3.5
```

There is also support for complex numbers, etc. Let's not get into that.

Strings

Python can also manipulate strings, which can be expressed in several ways. They can be enclosed in single quotes or double quotes:

>>> 'ha ha ha'

'ha ha ha'

>>> "ha ha ha "

'ha ha ha '

>>> """ ha ha ha """

' ha ha ha '

>>> 'doesn\'t'

"doesn't"

>>>

Strings can be concatenated (glued together) with the `+` operator, and repeated with `*`:

```
>>> word = 'Help' + 'A'
>>> word
'HelpA'
>>> '<' + word*5 + '>'
'<HelpAHelpAHelpAHelpAHelpA>'
>>> print 'ab' 'c'
abc
```

Strings can be subscripted (indexed); like in C, the first character of a string has subscript (index) 0. There is no separate character type; a character is simply a string of size one. Like in Icon, substrings can be specified with the *slice notation*: two indices separated by a colon.

```
>>> word[4]
'A'
>>> word[0:2]
'He'
>>> word[2:4]
'lp'
```

Slice indices have useful defaults; an omitted first index defaults to zero, an omitted second index defaults to the size of the string being sliced.

```
>>> word[:2]    # The first two characters
'He'
>>> word[2:]    # Everything except the first two characters
'lpA'
```

Unlike a C string, Python strings cannot be changed. Assigning to an indexed position in the string results in an error:

```
>>> word[0] = 'x'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object does not support item assignment
>>>
```

However, creating a new string with the combined content is easy and efficient:

```
>>> 'x' + word[1:]
'xelpA'
>>> 'Splat' + word[4]
'SplatA'
```

Here's a useful invariant of slice operations: `s[:i] + s[i:]` equals `s`.

```
>>> word[:2] + word[2:]
'HelpA'
>>> word[:3] + word[3:]
'HelpA'
```

Degenerate slice indices are handled gracefully: an index that is too large is replaced by the string size, an upper bound smaller than the lower bound returns an empty string.

Indices may be negative numbers, to start counting from the right. For example:

```
>>> word[-1]     # The last character
'A'
>>> word[-2]     # The last-but-one character
'p'
>>> word[-2:]    # The last two characters
'pA'
>>> word[:-2]    # Everything except the last two characters
'Hel'
```

But note that -0 is really the same as 0, so it does not count from the right!

```
>>> word[-0]     # (since -0 equals 0)
'H'
```

The built-in function `len()` returns the length of a string:

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

Lists¶

A *list* consists of a list of comma-separated values (items) between square brackets. List items need not all have the same type.

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
```

Like string indices, list indices start at 0, and lists can be sliced, concatenated and so on:

```
>>> a[0]
'spam'
>>> a[3]
1234
>>> a[-2]
100
>>> a[1:-1]
['eggs', 100]
>>> a[:2] + ['bacon', 2*2]
['spam', 'eggs', 'bacon', 4]
>>> 3*a[:3] + ['Boo!']
['spam', 'eggs', 100, 'spam', 'eggs', 100, 'spam', 'eggs', 100, 'Boo!']
```

All slice operations return a new list containing the requested elements. This means that the following slice returns a shallow copy of the list *a*:

```
>>> a[:]
['spam', 'eggs', 100, 1234]
```

Unlike strings, which are *immutable*, it is possible to change individual elements of a list:

```
>>> a
['spam', 'eggs', 100, 1234]
>>> a[2] = a[2] + 23
>>> a
['spam', 'eggs', 123, 1234]
```

Assignment to slices is also possible, and this can even change the size of the list or clear it entirely:

```
>>> # Replace some items:
... a[0:2] = [1, 12]
>>> a
[1, 12, 123, 1234]
>>> # Remove some:
... a[0:2] = []
>>> a
[123, 1234]
>>> # Insert some:
... a[1:1] = ['bletch', 'xyzzy']
>>> a
[123, 'bletch', 'xyzzy', 1234]
>>> # Insert (a copy of) itself at the beginning
>>> a[:0] = a
>>> a
[123, 'bletch', 'xyzzy', 1234, 123, 'bletch', 'xyzzy', 1234]
>>> # Clear the list: replace all items with an empty list
>>> a[:] = []
>>> a
[]
```

The built-in function **`len()`** also applies to lists:

```
>>> a = ['a', 'b', 'c', 'd']
>>> len(a)
4
```

It is possible to nest lists (create lists containing other lists), for example:

```
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> len(p)
3
>>> p[1]
[2, 3]
>>> p[1][0]
2
>>> p[1].append('xtra')     # See section 5.1
>>> p
[1, [2, 3, 'xtra'], 4]
>>> q
[2, 3, 'xtra']
```

First Steps Towards Programming¶

Of course, we can use Python for more complicated tasks than adding two and two together. For instance, we can write an initial sub-sequence of the *Fibonacci* series as follows:

```
>>> # Fibonacci series:
... # the sum of two elements defines the next
... a, b = 0, 1
>>> while b < 10:
...     print b
...     a, b = b, a+b
...
1
```

```
1
2
3
5
8
```

This example introduces several new features.

- The first line contains a *multiple assignment*:
- The `while` loop executes as long as the condition (here: `b < 10`) remains true. In Python, like in C, any non-zero integer value is true; zero is false.
- The *body* of the loop is *indented*: indentation is Python's way of grouping statements. Python does not (yet!) provide an intelligent input line editing facility, so you have to type a tab or space(s) for each indented line. In practice you will prepare more complicated input for Python with a text editor; most text editors have an auto-indent facility. \
- The `print` statement writes the value of the expression(s) it is given.

A trailing comma avoids the newline after the output:

```
>>> a, b = 0, 1
>>> while b < 1000:
...     print b,
...     a, b = b, a+b
...
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

Note that the interpreter inserts a newline before it prints the next prompt if the last line was not completed.

`if` Statements¶

```
>>> x = int(raw_input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...      x = 0
...      print 'Negative changed to zero'
... elif x == 0:
...      print 'Zero'
... elif x == 1:
...      print 'Single'
... else:
...      print 'More'
...
More
```

`for` Statements¶

The `for` statement in Python differs a bit from what you may be used to in C or Pascal. Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as C), Python's `for` statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. For example (no pun intended):

```
>>> # Measure some strings:
... a = ['cat', 'window', 'defenestrate']
>>> for x in a:
...     print x, len(x)
...
cat 3
window 6
defenestrate 12
```

The `range()` Function¶

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]
```

To iterate over the indices of a sequence, you can combine `range()` and `len()` as follows:

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print i, a[i]
...
0 Mary
1 had
2 a
3 little
4 lamb
```

`pass` Statements¶

The `pass` statement does nothing. It can be used when a statement is required syntactically but the program requires no action. For example:

```
>>> while True:
...     pass  # Busy-wait for keyboard interrupt (Ctrl+C)
...
```

Defining Functions¶

We can create a function that writes the Fibonacci series to an arbitrary boundary:

```
>>> def fib(n):    # write Fibonacci series up to n
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while a < n:
...         print a,
...         a, b = b, a+b
...
>>> # Now call the function we just defined:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

The keyword `def` introduces a function *definition*. It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented.

The first statement of the function body can optionally be a string literal; this string literal is the function's documentation string, or *docstring*. (More about docstrings can be found in the section *Documentation Strings*.) There are tools which use docstrings to automatically produce online or printed documentation, or to let the user interactively browse through code; it's good practice to include

docstrings in code that you write, so make a habit of it.

A function definition introduces the function name in the current symbol table. The value of the function name has a type that is recognized by the interpreter as a user-defined function. This value can be assigned to another name which can then also be used as a function. This serves as a general renaming mechanism:

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

Coming from other languages, you might object that `fib` is not a function but a procedure since it doesn't return a value. In fact, even functions without a **`return`** statement do return a value, albeit a rather boring one. This value is called **`None`** (it's a built-in name). Writing the value **`None`** is normally suppressed by the interpreter if it would be the only value written. You can see it if you really want to using **`print`**:

```
>>> fib(0)
>>> print fib(0)
None
```

It is simple to write a function that returns a list of the numbers of the Fibonacci series, instead of printing it:

```
>>> def fib2(n): # return Fibonacci series up to n
...     """Return a list containing the Fibonacci series up to n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a)    # see below
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)    # call it
>>> f100                # write the result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Data Structures

More on Lists¶

The list data type has some more methods. Here are all of the methods of list objects:

`list.`**`append`**(*x*)
Add an item to the end of the list; equivalent to `a[len(a):] = [x]`.
`list.`**`extend`**(*L*)
Extend the list by appending all the items in the given list; equivalent to `a[len(a):] = L`.
`list.`**`insert`**(*i*, *x*)
Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.
`list.`**`remove`**(*x*)
Remove the first item from the list whose value is *x*. It is an error if there is no such item.
`list.`**`pop`**([*i*])
Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. (The square brackets around the *i* in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)
`list.`**`index`**(*x*)
Return the index in the list of the first item whose value is *x*. It is an error if there is no such item.
`list.`**`count`**(*x*)
Return the number of times *x* appears in the list.
`list.`**`sort`**()
Sort the items of the list, in place.
`list.`**`reverse`**()

Reverse the elements of the list, in place.
An example that uses most of the list methods:

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print a.count(333), a.count(66.25), a.count('x')
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
```

Using Lists as Stacks¶

The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved ("last-in, first-out"). To add an item to the top of the stack, use **`append()`**. To retrieve an item from the top of the stack, use **`pop()`** without an explicit index. For example:

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

(probably skip) Map, reduce, filter, list comprehensions.

`filter(function, sequence)` returns a sequence consisting of those items from the sequence for which `function(item)` is true.

`map(function, sequence)` calls `function(item)` for each of the sequence's items and returns a list of the return values. For example, to compute some cubes:

`reduce(function, sequence)` returns a single value constructed by calling the binary function *function* on the first two items of the sequence, then on the result and the next item, and so on.

```
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [3*x for x in vec if x > 3]
[12, 18]
```

The **`del`** statement¶

There is a way to remove an item from a list given its index instead of its value: the **`del`** statement. This differs from the **`pop()`** method

which returns a value. The **del** statement can also be used to remove slices from a list or clear the entire list (which we did earlier by assignment of an empty list to the slice). For example:

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

del can also be used to delete entire variables:

```
>>> del a
```

Sets¶

Python also includes a data type for *sets*. A set is an unordered collection with no duplicate elements. Basic uses include membership testing and eliminating duplicate entries. Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.

Here is a brief demonstration:

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> fruit = set(basket)               # create a set without duplicates
>>> fruit
set(['orange', 'pear', 'apple', 'banana'])
>>> 'orange' in fruit                 # fast membership testing
True
>>> 'crabgrass' in fruit
False

>>> # Demonstrate set operations on unique letters from two words
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                 # unique letters in a
set(['a', 'r', 'b', 'c', 'd'])
>>> a - b                             # letters in a but not in b
set(['r', 'd', 'b'])
>>> a | b                             # letters in either a or b
set(['a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'])
>>> a & b                             # letters in both a and b
set(['a', 'c'])
>>> a ^ b                             # letters in a or b but not both
set(['r', 'd', 'b', 'm', 'z', 'l'])
```

Dictionaries¶

Another useful data type built into Python is the *dictionary*. Dictionaries are sometimes found in other languages as “associative memories” or “associative arrays”. Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by *keys*, which can be any immutable type; strings and numbers can always be keys. Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key. You can’t use lists as keys, since lists can be modified in place using index assignments, slice assignments, or methods like **`append()`** and **`extend()`**.

It is best to think of a dictionary as an unordered set of *key: value* pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary: `{}`. Placing a comma-separated list of key:value pairs within the braces adds initial key:value pairs to the dictionary; this is also the way dictionaries are written on output.

The main operations on a dictionary are storing a value with some key and extracting the value given the key. It is also possible to delete a key:value pair with `del`. If you store using a key that is already in use, the old value associated with that key is forgotten. It is an error to extract a value using a non-existent key.

The **keys()** method of a dictionary object returns a list of all the keys used in the dictionary, in arbitrary order (if you want it sorted, just apply the **sorted()** function to it). To check whether a single key is in the dictionary, use the **in** keyword.

Here is a small example using a dictionary:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> tel.keys()
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
```

The **dict()** constructor builds dictionaries directly from lists of key-value pairs stored as tuples.

Looping Techniques¶

When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the **iteritems()** method.

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.iteritems():
...     print k, v
...
gallahad the pure
robin the brave
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

Reading and Writing Files¶

open() returns a file object, and is most commonly used with two arguments: `open(filename, mode)`.

```
>>> f = open('/tmp/workfile', 'w')
>>> print f
<open file '/tmp/workfile', mode 'w' at 80a0960>
```

The first argument is a string containing the filename. The second argument is another string containing a few characters describing the way in which the file will be used. *mode* can be `'r'` when the file will only be read, `'w'` for only writing (an existing file with the same name will be erased), and `'a'` opens the file for appending; any data written to the file is automatically added to the end. `'r+'` opens the file for both reading and writing. The *mode* argument is optional; `'r'` will be assumed if it's omitted.

Methods of File Objects¶

The rest of the examples in this section will assume that a file object called `f` has already been created.

To read a file's contents, call `f.read(size)`, which reads some quantity of data and returns it as a string. *size* is an optional numeric argument. When *size* is omitted or negative, the entire contents of the file will be read and returned; it's your problem if the file is twice as large as your machine's memory. Otherwise, at most *size* bytes are read and returned. If the end of the file has been reached, `f.read()` will return an empty string (`""`).

```
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

`f.readline()` reads a single line from the file; a newline character (`\n`) is left at the end of the string, and is only omitted on the last line of the file if the file doesn't end in a newline. This makes the return value unambiguous; if `f.readline()` returns an empty string, the

end of the file has been reached, while a blank line is represented by `'\n'`, a string containing only a single newline.

```
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''
```

`f.readlines()` returns a list containing all the lines of data in the file. If given an optional parameter *sizehint*, it reads that many bytes from the file and enough more to complete a line, and returns the lines from that. This is often used to allow efficient reading of a large file by lines, but without having to load the entire file in memory. Only complete lines will be returned.

```
>>> f.readlines()
['This is the first line of the file.\n', 'Second line of the file\n']
```

An alternative approach to reading lines is to loop over the file object. This is memory efficient, fast, and leads to simpler code:

```
>>> for line in f:
        print line,

This is the first line of the file.
Second line of the file
```

When you're done with a file, call `f.close()` to close it and free up any system resources taken up by the open file.

also: f.seek(), f.tell()

---

lists of lists:

```
>>> a = [1, 2, 3, 4]
>>> a[1]
2
>>> a[1]=10
>>> a
[1, 10, 3, 4]
>>> len(a)
4
>>> b=[a,4,5,7,a]
>>> b
[[1, 10, 3, 4], 4, 5, 7, [1, 10, 3, 4]]
>>> b[0].append(100)
>>> b
[[1, 10, 3, 4, 100], 4, 5, 7, [1, 10, 3, 4, 100], 'xtra']
>>> a
[1, 10, 3, 4, 100]
```

working with a file test.txt that contains:

1 3 5 6 7

8 9 10

```
>>> f=open('/Users/ltoma/Desktop/test.txt')
>>> f.read()
'1 3 5 6 7\n8 9 10 \n'
>>> f.readline()
''
>>> f.seek(0)
>>> f.readline()
'1 3 5 6 7\n'
```

```
>>> f.seek(0)
>>> for line in f:  print line,
...
1 3 5 6 7
8 9 10
>>> f.seek(0)
>>> f.readlines()
['1 3 5 6 7\n', '8 9 10 \n']
l = f.readline()
>>> print l
1 3 5 6 7
>>> for x in l: print x
...
 1
 3
 5
 6
 7
>>> for x in f: print x
...
1 3 5 6 7
8 9 10
>>> f.seek(0)
>>> l = f.readline()
>>> print l
1 3 5 6 7

### l is a string. How to make a list from l?
>>>l.strip()  ##to get rid of the \n
>>> l=l.strip()
>>> l
'1 3 5 6 7'
>>> l.split()    #this creates a list from a string by splitting it into tokens
['1', '3', '5', '6', '7']
```

Let's take a different type of file. cast.06.txt: movies released in 2006 [movies=8780, actors=84236]

Each line gives the name of a movie followed by the cast. Since names have spaces and commas in them, the / character is used as a delimiter.

```
'Breaker' Morant (1980)/Fitz-Gerald, Lewis/Steele, Rob (I)/Wilson, Frank (II)/Tingwell, Charles 'Bud'/
Cassell, Alan (I)/Rodger, Ron/Knez, Bruno/Woodward, Edward/Cisse, Halifa/Quin, Don/Kiefel, Russell/Meagher,
Ray/Procanin, Michael/Bernard, Hank/Gray, Ian (I)/Brown, Bryan (I)/Ball, Ray (I)/Mullinar, Rod/Donovan,
Terence (I)/Ball, Vincent (I)/Pfitzner, John/Currer, Norman/Thompson, Jack (I)/Nicholls, Jon/Haywood, Chris
(I)/Smith, Chris (I)/Mann, Trevor (I)/Henderson, Dick (II)/Lovett, Alan/Bell, Wayne (I)/Waters, John (III)/
Osborn, Peter/Peterson, Ron/Cornish, Bridget/Horseman, Sylvia/Seidel, Nellie/West, Barbara/Radford, Elspeth/
Reed, Maria/Erskine, Ria/Dick, Judy/Walton, Laurie (I)
...
```

Let's try to read this file in Python, and figure out the movie and its actors.

```
>>> f=open('cast.06.txt')
>>> f.tell()
0L
>>> f.readline()
'"Tis Autumn: The Search for Jackie Paris (2006)/Paris, Jackie/Vera, Billy/Bogdanovich, Peter/Newman, Barry (I)/Ellison, Harlan/Tosches, Nick (I)/Moody,
James (IV)/Whaley, Frank/Murphy, Mark (X)/Moss, Anne Marie\n'
>>> l=f.readline()
>>> l
'(Desire) (2006)/Ruggieri, Elio/Ferdinando, Emma/Micijevic, Irena/Sweeney, Scarlet/Nesbitt, Beryl\n'
>>> l[0]
'('
>>>
>>> l.strip()
'(Desire) (2006)/Ruggieri, Elio/Ferdinando, Emma/Micijevic, Irena/Sweeney, Scarlet/Nesbitt, Beryl'
>>> l.split()
['(Desire)', '(2006)/Ruggieri,', 'Elio/Ferdinando,', 'Emma/Micijevic,', 'Irena/Sweeney,', 'Scarlet/Nesbitt,', 'Beryl']
>>> ll=l.split()
>>> for x in ll: print x
...
##we need  to split with / as separator.
>>> l.split('/')
['(Desire) (2006)', 'Ruggieri, Elio', 'Ferdinando, Emma', 'Micijevic, Irena', 'Sweeney, Scarlet', 'Nesbitt, Beryl\n']
>>> ll=l.split('/')
>>> ll
```

```
['(Desire) (2006)', 'Ruggieri, Elio', 'Ferdinando, Emma', 'Micijevic, Irena', 'Sweeney, Scarlet', 'Nesbitt, Beryl\n']
>>> for x in ll: print x
...
(Desire) (2006)
Ruggieri, Elio
Ferdinando, Emma
Micijevic, Irena
Sweeney, Scarlet
Nesbitt, Beryl
```

How to get the movie name?

```
>>> ll[0]
'(Desire) (2006)'
>>> ll[0].split()
['(Desire)', '(2006)']
>>> movie=ll[0].split()[0]
>>> movie
'(Desire)'
```

How to get rid of parenthesis?

```
>>> year=ll[0].split()[1]
>>> year
'(2006)'
>>> movie[1:]
'Desire)'
>>> movie=movie[1:]
>>> movie
'Desire)'
>>> movie[:-1]
'Desire'
>>> movie=movie[:-1]
>>> movie
'Desire'
##let's say we keep a global  list of movies
>>> mymovies=[]
>>> mymovies.append(movie)
>>> mymovies
['Desire']
```

Now we'd like to read the set of actors in the movie, and record them. For each movie, we are going to store the list of its actors, in pairs [movie, actor-list]. We are going to store this in a dictionary, with key=movie. remember that a dictionary is a set of key-value pairs. For us, the key is the movie name, and the value is the list of actors in the movie.

```
>>>actors= ['a1','a2','a3','a4']
>>> dict({movie:actors})
{'Desire': ['a1', 'a2', 'a3', 'a4']}
>>> db = dict({movie:actors})
>>> db
{'Desire': ['a1', 'a2', 'a3', 'a4']}
##adding items to the dictionary. Read the dictionary interface.
>>> db.setdefault('travel',['b1','b2','b3'])
['b1', 'b2', 'b3']
>>> db
{'Desire': ['a1', 'a2', 'a3', 'a4'], 'travel': ['b1', 'b2', 'b3']}

##method keys() which returns all keys
>>> db.keys()
['Desire', 'travel']
##method items() returns the key-value pairs
>>db.items()
>>> for x in db.items(): print x
...
('Desire', ['a1', 'a2', 'a3', 'a4'])
('travel', ['b1', 'b2', 'b3'])
>>> for a,b in db.items(): print a,b
...
Desire ['a1', 'a2', 'a3', 'a4']
travel ['b1', 'b2', 'b3']
```

Now, we can do stuff like this: (again, check teh dictionary interface in Python)

```
>>> db['Desire']
['a1', 'a2', 'a3', 'x', 'a5']
```

```
##this immediately gives us the actors in that movie
```

EXERCISE 1: Read the entire file and store the movies and actors in the dictionary as above. At the end print the number of movies. And print the movies in sorted order.

EXERCISE 2: Suppose we want a list of all actors in all the movies, and for each actor we want to know all the movies he/she played in. Construct such a list. Note that it will be a symmetrical structure, a list of actor,movies pairs.

There are several ways you could do this. You could do it as you read lines from the file, in the same time as you build the movie,actors structure. Note also that you could store this as a second dictionary, or in the same dictionary. Basically, the dictionary would store(string, list of strings) pairs, where the string can be either a movie name or an actor name.

Or, you could first build the (movie,actors) dictionary, and then you could iterate over iit and figure out the actors.

Example of iterators over the dictionary:

```
>>>for a in db.keys()   ## this iterates over all the movies
>>>...  db.get(a)

>>> for a,b in db.items():  ##a is the movie name and b is the list of actors
...      for c in b:  ##now we iterate over b
...              print c
...  ##this will print all the actors in all the movies. Note that this is not a set.
## we could make a set like this:
set(the list of all actors)
```