Appendix A: Command Overview

A.1 UNIX COMMANDS

A.1.1 Listing Files and Directories

`ls`	Lists files and directories
`ls -a`	Lists hidden files and directories
`mkdir`	Creates a directory
`cd directory`	Changes to named directory
`cd`	Changes to home directory
`cd ~`	Changes to home directory
`cd..`	Changes to parent directory
`pwd`	Displays the path of the current directory

A.1.2 Handling Files and Directories

`cp file1 file2`	Copies file1 and calls it file2
`mv file1 file2`	Moves or renames file1 to file2
`rm file`	Removes a file
`rmdir directory`	Removes a directory
`cat file`	Displays a file
`more file`	Displays a file a page at a time
`head file`	Displays the first few lines of a file
`tail file`	Displays the last few lines of a file
`grep 'keyword' file`	Searches a file for keywords
`wc file`	Counts the number of lines, words, and characters in a file

A.1.3 Redirection

`command > file`	Redirects standard output to a file
`command >> file`	Appends standard output to a file
`command < file`	Redirects standard input from a file
`cat file1 file2 > file0`	Concatenates file1 and file2 to file0
`sort`	Sorts data
`who`	List users currently logged in

A.1.4 File System Security (Access Rights)

`ls -lag`	Lists access rights for all files
`chmod [options] file`	Changes access rights for named file
`command &`	Runs command in background
`^C`	Kills the job running in the foreground
`^Z`	Suspends the job running in the foreground
`bg`	Backgrounds the suspended job
`jobs`	Lists current jobs
`fg%1`	Foreground job number 1
`kill%1`	Kills job number 1
`ps`	Lists current processes
`kill 26152`	Kills process number 26152

A.1.5 `chmod` Options

Symbol	Meaning
`u`	User
`g`	Group
`o`	Other
`a`	All
`r`	Read
`w`	Write (and delete)
`x`	Execute (and access directory)
`+`	Add permission
`-`	Take away permission

A.1.6 General Rules

- If you've made a typo, use Ctrl-U to cancel the whole line.
- UNIX is case-sensitive.

- There are commands that can take options.
- The options change the behavior of the command.
- UNIX uses command-line completion.
- `%command_name -options <file> [Return]`
- `%man <command name> [Enter]`
- `%whatis <command name> [Enter]`
- Ctrl-A sets the cursor at the beginning of the line.
- Ctrl-E sets the cursor at the end of the line.
- You can use up and down arrows to recall commands.
- The command `whereis` tells you where a given program is.
- You can use a text editor to write stuff (e.g., gedit).

A.2 PYTHON COMMANDS

A.2.1 Overview

- Python is an interpreted language.
- The Python interpreter automatically generates bytecode (`.pyc` files).
- It is 100% free software.

Strengths

- It is quick to write, and there is no compilation.
- It is fully object oriented.
- It has many reliable libraries.
- It is an all-round language.

Weakness

- Writing very fast programs is not easy.

A.2.2 The Python Shell

Overview

You can use any Python command from the interactive Python shell (>>>):

```
>>> print 4**2
16
>>> a = 'blue'
>>> print a
blue
>>>
```

Tips

- All Python commands work in the same way in the Python shell and in programs.
- You can leave the command line by Ctrl-D (Linux, Mac) or Ctrl-Z (Windows).
- The Python shell works great as a pocket calculator.
- Writing code blocks with more than two lines in the Python shell gets painful quickly.
- You can define code blocks by writing extra lines:

```
>>> for i in range(3):
...       print i,
...       012
>>>
```

A.2.3 Python Programs

- All program files should have the extension .py.
- Each line should contain exactly one command.
- Code blocks are marked by indentation. Code blocks should be indented by four spaces or one tab.
- When you are developing on UNIX, the first line in each Python program should be

```
#!/usr/bin/env python
```

Code Formatting Conventions

- Use spaces instead of tabs (or use an editor that converts them automatically).
- Keep lines shorter than 80 characters long.
- Separate functions with two blank lines.
- Separate logical chunks of long functions with a single blank line.
- Variables and function names are in lowercase.

The Dogma of Programming

- First, make it work.
- Second, make it nice.
- Third, and only if it is really necessary, make it fast.

A.2.4 Operators

Arithmetic Operators

`7 + 4`	Addition; results in 11 (works for strings and lists)
`7 - 4`	Subtraction; results in 3
`7 * 4`	Multiplication; results in 28
`7 / 4`	Division of integers; results in 1 (rounded down)
`7 / 4.0`	Division by float; results in 1.75
`7 % 4`	Modulo operator, returns the remainder of a division; results in 3
`7 ** 2`	Raising to a power; results in 49
`7.0 // 4.0`	Floor division (cuts off after point); results in 1.0

Assignment Operators

Assignment operators create or modify variables.

Variables

Variable are containers for data. Variable names may be composed of letters, underscores, and, after the first position, digits. Lowercase letters are common, but uppercase is also allowed (usually used for constants). Some words such as `print`, `import`, and `for` are forbidden as variable names.

`a = 10`	Assigns the integer value 10 to the variable `a`
`b = 3.0`	Variable containing a floating-point number
`a3 = 10`	Variable with a digit in its name
`PI = 3.1415`	Variable written with uppercase letters
`invitation = 'Hello World'`	Variable containing text (string)
`invitation = "Hello World"`	Variable containing text with double quotes

Modifying Variables

`+=, -=, *=,`	Recursive operators
`/=,%=, **=`	`x += 1` is equivalent to `x = x+1`

Comparison Operators

All comparison variables result in either `True` or `False`.

`a == b`	`a` equal to `b`
`a != b`	`a` not equal to `b`
`a < b`	`a` smaller than `b`
`a > b`	`a` bigger than `b`
`a <= b`	`a` smaller or equal to `b`
`a >= b`	`a` smaller or equal to `b`
`in, not in`	The `in` and `not in` operators check if the object on their left is contained (or not contained) in the string, list, or dictionary on their right and return the Boolean value `True` or `False`.
`is, is not`	The `is` and `is not` operators check whether the object on their left is identical (or not identical) to the object on their right and return the Boolean value `True` or `False`.
`a and b`	Boolean operator: if both the condition `a` and `b` are `True`, it returns `True`; else it returns `False`.
`or`	Boolean operator: if either the condition `a` or `b` or both are `True`, it returns `True`; else it returns `False`.
`not a`	Boolean operator: if the condition `a` is not verified, it returns `True`; else it returns `False`.

A.2.5 Data Structures

Overview of Data Types

Integers	Are numbers without digits after the decimal point
Floats	Are numbers with digits after the decimal point
Strings	Are *immutable ordered* collections of characters and are indicated with single (`'abc'`) or double (`"abc"`) quotation marks
Lists	Are *mutable ordered* collections of objects and are indicated with square brackets (`[a,b,c]`)
Tuples	Are *immutable ordered* collections of objects and are indicated with round brackets (`(a,b,c)`) or by listing the collection of items separated by commas (`x, y, z`)
Dictionaries	Are unordered collections of `key:value` pairs
Sets	Are collections of unique elements
Boolean	Are either `True` or `False`

Type Conversions

`int(value)`	Creates an integer from a float or string
`float(value)`	Creates a float from an `int` or string
`str(value)`	Creates a string from any variable

A.2.6 Strings

String variables are containers for text. Strings can be marked by many kinds of quotes, which are all equivalent.

`s = 'Hello World'`	Assigns the text to a string variable
`s = "Hello World"`	String with double quotes
`s = '''Hello World'''`	Multiline string with triple single quotes
`s = """Hello World"""`	Multiline string with triple double quotes
`s = 'Hello\tWorld\n'`	String with a tab (`\t`) and a newline (`\n`)

Accessing Characters and Substrings

Using square brackets, any character of a string can be accessed by its position. The first character has the index 0. Substrings can be formed by square brackets with two numbers separated by a colon. The position corresponding to the second number is not included in the substring. If you try to use indices bigger than the length of the string, an `IndexError` will be created.

`print s[0]`	Prints the first character
`print s[3]`	Prints the fourth character
`print s[-1]`	Prints the last character
`print s[1:4]`	Second to fourth position; results in `'ell'`
`print s[:5]`	From start to fifth position; results in `'Hello'`
`print s[-5:]`	Fifth position from the end until the end; results in `'World'`

String Functions

A number of functions can be used on every string variable:

`len(s)`	Length of the string; results in `11`
`s.upper()`	Converts to uppercase; results in `'HELLO WORLD'`
`s.lower()`	Converts to lowercase; results in `'hello world'`
`s.strip()`	Removes spaces and tabs from both ends
`s.split(' ')`	Cuts into words; results in `['Hello', 'World']`
`s.find('llo')`	Searches for a substring; returns starting position
`s.replace('World','Moon')`	Replaces text; results in `'Hello Moon'`
`s.startwith('Hello')`	Checks beginning; returns `True` or `False`
`s.endswith('World')`	Checks end; returns `True` or `False`

A.2.7 Lists

A list is a sequence of elements that can be modified. In many cases, all elements of a list will have the same type, but this is not mandatory.

Accessing Elements of Lists

When you use square brackets, any element of a list can be accessed. The first character has the index 0.

`data = [1,2,3,4,5]`	Creates a list
`data[0]`	Accesses the first element
`data[3]`	Accesses the fourth element
`data[-1]`	Accesses the last element
`data[0] = 7`	Reassigns the first element

Creating Lists from Other Lists

You can extract sublists from lists by applying square brackets in the same way as you would extract substrings.

`data = [1,2,3,4,5]`	Creates a list
`data[1:3]`	`[2,3]`
`data[0:2]`	`[1,2]`
`data[:3]`	`[1,2,3]`
`data[-2:]`	`[4,5]`
`backup = data[:]`	Creates a copy of the list

Modifying Lists

`l[i] = x`	The `i`th element of `l` is replaced by `x`.
`l[i:j] = t`	The elements of `l` from `i` to `j` are replaced by `t` (iterable).
`del l[i:j]`	This deletes the elements of `l` from `i` to `j`.
`l[i:j:k] = t`	The elements `l[i:j:k]` are replaced by the elements of `t` (`t` must be a sequence such that `len(l[i:j:k]) = len(t)`).
`del s[i:j:k]`	This deletes the elements of `l` from `i` to `j` with step `k`.
`l.append(x)`	This is the same as `l[len(l):len(l)] = [x]`. It appends the element `x` to the list `l`.
`l.extend(x)`	This is the same as `l[len(l):len(l)] = x` (where `x` is any iterable object).
`l.count(x)`	This returns the number of elements `x` in `l`.
`l.index(x[, i[, j]])`	This returns the smaller `k` such that `l[k] = x` and `i ≤ k ≤ j`.
`l.insert(i, x)`	This is the same as `l[i:i] = [x]`.
`l.pop(i)`	This cancels the `i`th element and returns its value. `l.pop()` is the same as `del l[-1]; return l[-1]`.
`l.remove(x)`	This is the same as `del l[l.index(x)]`.
`l.reverse()`	This reverses the elements of `l`.
`l.sort()`	This sorts the list `l`.
`l.sort([cmp[, key[, reverse]]])`	This sorts the list `l`. Optional arguments for the control of the comparison can be passed to the `sort()` method. `cmp` is a customized function for the comparison of element pairs that must return a negative value, zero, or a positive value depending on if the first element of the pair is lower than, equal to, or greater than the second element.
`sorted(l)`	This creates a new list made of a simple ascending sort of `l` (without modifying `l`).

Functions Working on Lists

`data = [3,2,1,5]`	Example data
`len(data)`	Length of data; returns 4; also works for many other types
`min(data)`	Smallest element of data; returns 1
`max(data)`	Biggest element of data; returns 5
`sum(data)`	Sum of data; returns 11
`range(4)`	Creates a list of numbers; returns `[0,1,2,3]`
`range(1,5)`	Creates a list with start value; returns `[1,2,3,4]`
`range(2,9,2)`	Creates a list with step size; returns `[2,4,6,8]`
`range(5,0,-1)`	Creates a list counting backward from the start value; returns `[5,4,3,2,1]`

A.2.8 Tuples

A tuple is a sequence of elements that cannot be modified. This means that once you have defined it, you cannot change or replace its elements. They are useful to group elements of different types.

```
t = ('bananas','200g',0.55)
```

Notice that brackets are optional; i.e., you can use either `Tuple = (1,2,3)` or `Tuple = 1,2,3`.

A tuple of a single item must be written either `Tuple = (1,)` or `Tuple = 1`.

You can use square brackets to address elements of tuples in the same way as you would address elements of lists.

A.2.9 Dictionaries

Dictionaries are an unordered, associative array. They have a set of `key:value` pairs:

```
prices = {'banana':0.75, 'apple':0.55, 'orange':0.80}
```

In the example, `'banana'` is a key, and `0.75` is a value.

Dictionaries can be used to look up things quickly:

```
prices['banana'] # 0.75
prices['kiwi'] # KeyError
```

Accessing Data in Dictionaries

By applying square brackets with a key inside, you can request the values of a dictionary. Keys can be strings, integers, floats, and tuples.

`prices['banana']`	This returns the value of `'banana'` (`0.75`).
`prices.get('banana')`	This returns the value of `'banana'` but avoids the `KeyError`. If the key does not exist, it returns `None`.
`prices.has_key('apple')`	This checks whether `'apple'` is defined.
`prices.keys()`	This returns a list of all keys.
`prices.values()`	This returns a list of all values.
`prices.items()`	This returns all keys and values as a list of tuples.

Modifying Dictionaries

`prices['kiwi'] = 0.6`	Sets the value of `'kiwi'`
`prices.setdefault('egg',0.9)`	Sets the value of `'egg'` if it is not defined yet

The `None` Type

Variables can contain the value `None`. You can also use `None` to indicate that a variable is empty. `None` is also used automatically when a function does not have a return statement.

```
traffic_light = [None, None, 'green']
```

A.2.10 Control Flow

Code Blocks/Indentation

After any statement ending with a colon (:), all indented commands are treated as a code block and are executed within the loop `if` if the condition applies. The next unindented command marks the end of the code block.

Loops with `for`

Loops repeat commands. They require a sequence of items that they iterate, e.g., a string, list, tuple, or dictionary. Lists are useful when you know the number of iterations in advance and when you want to do the same thing to all elements of a list.

`for base in 'AGCT':` `  print base`	Prints four lines containing A, G, C, and T
`for number in range(5):` `  print number`	Prints five lines with the numbers from 0 to 4
`for elem in [1, 4, 9, 16]:` `  print elem`	Prints the four numbers each to a separate line

Counting through Elements of Lists

The `enumerate()` function associates an integer number starting from zero to each element in a list. This is helpful in loops where an index variable is required.

```
>>> fruits = ['apple','banana','orange']
>>> for i, fruit in enumerate(fruits):
...     print i, fruit
...
0 apple
1 banana
2 orange
```

Merging Two Lists

The `zip()` function associates the elements of two lists to a single list of tuples. Excess elements are ignored.

```
>>> fruits = ['apple','banana','orange']
>>> prices = [0.55, 0.75, 0.80, 1.23]
>>> for fruit,price in zip(fruits,prices):
...     print fruit, price
...
apple 0.55
banana 0.75
orange 0.8
```

Loops over a Dictionary

You can access the keys of a dictionary in a `for` loop. However, their order is not guaranteed.

Conditional Statements with `if`

`if` statements are used to implement decisions and branching in the program. They must contain an `if` block and optionally one or many `elif` and `else` blocks:

```
if fruit == 'apple':
   price = 0.55
elif fruit == 'banana':
   price = 0.75
elif fruit == 'orange':
   price = 0.80
else:
   print 'we dont have%s'%(fruit)
```

Comparison Operators

An expression with `if` may contain any combination of comparison operators, variables, numbers, and function calls:

- `a == b, a != b` (equality)
- `a < b, a > b, a <= b, a >= b` (relations)
- `a or b, a and b, not a` (Boolean logic)
- `(a or b) and (c or d)` (priority)
- `a in b` (inclusion, when `b` is a list, tuple, or string)

Boolean Value of Variables

Apart from the comparison operators, the `if` statement also takes the values of variables directly into account. Each variable can be interpreted by Boolean logic. All variables are `True`, except for

```
0, 0.0, '', [], {}, False, None
```

Conditional Loops with `while`

`while` loops require a conditional expression at the beginning. These work in exactly the same way as in `if...elif` statements:

```
>>> i = 0
>>> while i < 5:
...     print i,
...     i = i + 1
...
0 1 2 3 4
```

When to Use `while`

- When there is a loop exit condition
- When you want to start a loop only upon a given condition
- When it may happen that nothing is done at all
- When the number of repeats depends on user input
- When you are searching for a particular element in a list

A.2.11 Program Structures

Functions

Functions are subprograms. They help you to structure your code into logical units. A function may have its own variables. It also has an input (parameters) and output (returned values). In Python, a function is defined by the `def` statement, followed by the function name, the argument(s) in brackets, a colon (:), and an indented code block:

```
def calc_discount(fruit, n):
    '''Returns a lower price of a fruit.'''
    print 'Today we have a special offer for:', fruit
    return 0.75 * n
print calc_discount('banana', 10)
```

Parameters and Return Values

Input for a function is given by arguments. Arguments may have default values. Then they are optional in the function call. *Do not use lists or dictionaries as default values!*

The output of a function is created by the `return` statement. The value given to `return` goes to the program part that called it. More than one value is returned as a tuple. In any case, the `return` statement ends the function execution.

```
def calc_disc(fruit,n = 1):        # A function with an optional
    print fruit                    # parameter
    return n*0.75

def calc_disc(fruit,n = 1):        # A function returning a tuple
    fruit = fruit.upper()
    return fruit, n*0.75
calc_disc('banana')                # Function calls
calc_disc('banana',100)
```

Good Style for Writing Functions

- Each function should have one purpose only.
- The name should be clear and start with a verb.
- The function should have a triple-quoted comment at the beginning (a documentation string).
- The function should return results in only one way.
- Functions should be small (fewer than 100 lines).

A.2.12 Modules

A module is a Python file (the filename ending with `.py`). Modules can be imported from another Python program. When a module is imported, the code within is automatically executed. To import from a module, you need to give its name (without `.py`) in the `import` statement. It is helpful to explicitly list the variables and functions required. This helps with debugging.

`import math`	Includes and interprets a module
`from math import sqrt`	Includes one function from a module
`from math import pi`	Includes a variable from a module
`from math import sqrt, pi`	Includes both
`from math import *`	Includes everything from a module (merges namespaces)

Finding Out What Is in a Module

The contents of any module can be examined with `dir()` and `help()`.

`dir(math)`	Shows everything inside the module
`dir()`	Shows everything in the global namespace
`help(math.sqrt)`	Displays the help text of a module or function
`__name__`	Name of a module
`__doc__`	Help text of a module
`__builtins__`	Container with all standard Python functions

Where Python Looks for Modules

When importing modules or packages, Python looks in

- the current directory,
- the `Python2.6/lib/site-packages` folder, and

- everything in the PYTHONPATH environment variable. In Python, it can be accessed with

```
import sys
print sys.path
sys.path.append('my_directory')
import my_package.my_module
```

Packages

For very big programs, you might find it useful to divide the Python code into several directories. There are two things to keep in mind when doing that:

- To import the package from outside, you need to ensure a file __init__.py (it may be empty) is in the package directory.
- The directory with the package needs to be in the Python search path (see above).

A.2.13 Input and Output

Reading Text from the Keyboard into a Variable

User input can be read from the keyboard with or without a message text:

`a = raw_input()`	Reads text from the keyboard to a string variable
`a = raw_input('please enter a number')`	Displays the text, then reads a string from the keyboard

Printing Text

The Python `print` statement writes text to the console where Python was started. The `print` command is very versatile and accepts almost any combination of strings, numbers, function calls, and arithmetic operations separated by commas. By default, `print` generates a newline character at the end.

`print 'Hello World'`	Displays a text
`print 3.4`	Displays the number
`print 3 + 4`	Displays the result of the calculation
`print a`	Displays the contents of the variable `a`
`print '''line one line two line three'''`	Displays text stretching over multiple lines
`print 'number', 77`	Displays the text, a tab, and the number

`print int(a) * 7`	Displays the result of the multiplication after converting the variable to an integer
`print`	Displays an empty line

String Formatting

Variables and strings can be combined using formatting characters. This works also within a `print` statement. In both cases, the number of values and formatting characters must be equal.

```
s = 'Result:%i'%(number)
print 'Hello%s!'%('Roger')
print '(%6.3f/%6.3f)'%(a,b)
```

The formatting characters include the following:

- `%i`: an integer
- `%4i`: an integer formatted to length 4
- `%6.2f`: a float number with length 6 and 2 after the comma
- `%10s`: a right-oriented string with length 10

Reading and Writing Files

Text files can be accessed using the `open()` function. It returns an open file whose contents can be extracted as a string or strings that can be written. If you try to open a file that does not exist, an `IOError` will be created.

`f = open('my_file.txt')` `text = f.read()`	Reads a text file and its contents into a string variable
`f = open('my_file.txt','w')` `f.write(text)`	Creates a new text file and writes text from a string variable into it
`f = open('my_file.txt','a')` `f.write(text)`	Appends text to an already existing file
`f.close()`	Closes a file after usage; closing in Python is good style but not always mandatory
`lines = f.readlines()`	Reads all lines from a text file to a list
`f.writelines(lines)`	Writes a list of lines to a file
`for line in open(name):` `  print line`	Goes through all lines and prints each line

`lines = ['first line\n', 'second line\n']`
`f = open('my_file.txt','w')`
`f.writelines(lines)`
	Creates a list of lines with newline characters at the end and saves it to a text file