MASSIMO DI PIERRO

WEB AUTOMATION WITH PYTHON

EXPERTS4SOLUTIONS

For more information about appropriate use of this material contact:

```
Massimo Di Pierro
School of Computing
DePaul University
243 S Wabash Ave
Chicago, IL 60604 (USA)
Email: massimo.dipierro@gmail.com
```

Library of Congress Cataloging-in-Publication Data:

ISBN: XXX-XXX-XXXX

Build Date: September 11, 2013

XXXXXXXXXXX

Contents

Preface

1 Introduction

Your account:

```
ssh user<yourid>@140.192.30.237
```

2
Linux

2.1 Access

In this course you will be using a Linux. There are many ways to use Linux. You can install on your machine with or without dual boot (dual boot is only recommended for experts), you run it within a virtual machine (VMWare or VirtualBox for example), or you can access a remote Linux server.

Assuming you have a Linux account on a remote server you can connect to it using one of the following ways:

- From another Linux/Unix/Mac terminal:

  ```
  ssh <username>@<hostname>
  ```

  for example:

  ```
  ssh thomas.anderson@the.matrix.net
  ```

- From Mac you can follow the same direction as Linux but first you have to open Terminal. Search for “Terminal” in “Spotlight”.

- From Windows you need to download PuTTy from `http://the.earth.li/~sgtatham/putty/latest/x86/putty.exe` and use the PuTTY GUI to insert the username and hostname.

- If you have a PythonAnywhere.com account, login there the click on [Consoles] and [Start a new console][Bash]. PythonAnywhere provides a free Linux account in the could and they offer a Bash console shell you can access via the web browser.

2.2 Bash Shell

The "Bash shell" is a command line processor. It allows you to interactively type and execute commands and runs inside a text window. There are many different shells, a.k.a. command line processors and they have a different syntax. For example the Bash shell understand "bash" commands; the Python shell understand "Python" commands. Even Windows has a shell (the "Command Prompt") which understand Windows commands. For now you will be using the Bash shell.

Example of commands:

What is my name?

```
whom
```

Make a file called "test.py" and write "print 1" in it:

```
echo "print 1" > test.py
```

List all files in the current folder

```
ls *
```

(∗ means everything and it is referred to as a wildcard). List all files ending in .py

```
ls *.py
```

List all files ending in .py with more details

```
ls -l *.py
```

Run the Python program test.py

```
python test.py
```

Delete all files ending in

```
rm *~
```

List the name of the current folder

```
pwd
```

Navigate to the parent of the current folder

```
cd ..
```

Navigate to the top folder (also known as file system root)

```
cd /
```

Navigate back to your home folder

```
cd ~
```

Make a new subfolder

```
mkdir work
```

Enter the subfolder

```
cd work
```

Create an empty file called “test.txt”

```
touch test.txt
```

Step out of a folder

```
cd ..
```

Remove a folder and all its content

```
rm -r work
```

Print a list of all programs I am running

```
ps
```

Print a list of all programs running on this machine by or other user

```
ps -aux
```

List all running programs and sort them by resource consumption

```
top
```

(press “q” to quit) Start a program and send it in background

```
python &
```

(the program will be running but will not IO to console) Kill a program currently running

```
kill -9 <pid>
```

(here `<pid>` is the process id of the program as listed by top and or ps).

Practice with the above commands.

`pwd`, `ls`, `cd`, `mkdir`, `echo`, `touch`, `rm`, `ps`, `top`.

Use them to explore the file system.

Also notice that you can type and incomplete command and press [tab], the Bash shell will complete the command for you or give you a list of options. This is called autocomplete.

Some operations require superuser access and the system will refuse to perform the action otherwise. If you are administrator of a system you can perform an action with administrator by prepending `sudo` to any shell command.

For example - DO NOT DO THIS, EVER! - delete everything in the file system:

```
cd /
sudo rm -r *
```

At this point you have the ability to create and browser files, folders and running processes. A running program may be constituted one or more processes (parallel programs may do more than one thing at once, thus more than one process).

2.3 Piping stdin and stdout

When using the `echo` command we used the following example:

```
echo "print 1" > test.py
```

Here > is a redirection operator. It takes the output of the `echo` command which would normally print to console, and send it to the `test.py` file. Similarly one can redirect standard input which is normally received from the keyboard and read is instead from a file using the < operator.

2.4 Installing prepackaged apps

At this point we need to to learn how to install new programs and run them. How you do this depends on the version of Linux. I will assume Ubuntu. I will also assume you have superuser access else you will not be able to install programs.

The easiest way to install prepackaged apps for Ubuntu is:

```
sudo apt-get <appname>
```

For example

```
sudo apt-get install mc
```

Once installed we can start it with

```
mc
```

`mc` is a useful extension to the Bash shell which allows us to type Bash command and at the same time to browse the file system in a visual way.

We can also re-install existing apps to make sure we have the latest version, for example:

```
sudo apt-get install python
```

Other useful apps are the following

```
sudo apt-get install zip    # for compressing files
sudo apt-get install unzip  # for uncompressing
sudo apt-get install tar    # for packing files
sudo apt-get install emacs  # a file editor
sudo apt-get install python # the python interpreter
sudo apt-get install g++    # the C++ compiler
sudo apt-get install wget   # to download stuff from web
sudo apt-get install make   # to install source packages
sudo apt-get install build-essential
sudo apt-get install python2.7  # to make sure we have it
sudo apt-get install python-dev # useful development tools
sudo apt-get install git        # version control system
```

2.5 Emacs in a nutshell

Emacs is a text editor. It understand the syntax of many programming languages and can help with development. It does not have a GUI but only a text based UI and you can perform various operations using command line shortcuts. Remember the most important of them all `[crtl]+g`. if you get into trouble it brings you back to the edit interface.

You can open a file with:

```
emacs test.py
```

`[crtl]+x+c`	Save and Exit
`[crtl]+s`	Search
`[meta]+<`	scroll to beginning
`[meta]+>`	scroll to end
`[meta]+X`	command prompt
`[tab]`	automatically re-indent the current line

2.6 Installing binary packages

Some time you may need to install binary packages. They come in the form of of tar g-zipped files. They look like: `<appname>.tar.gz`

They installed using the following process:

```
gunzip <appname>.tar.gz
tar xvf <appname>.tar
cd <appname-folder>
./configure
make
sudo make install
```

Here <appname> is the name of the application being installed. <appname-folder> is the folder created by `tar xvf ...`. The `tar` command extracts the content of the `.tar` file into a folder. The `./configure` step can take additional command line options and one should consult the documentation of the specific package for more details. `make` compiles the application. `make install` moves the compiled files in the proper place. If everything worked fine you can delete the `<appname-folder>`.

2.7 Git Tutorial

(from the Git man pages)

This tutorial explains how to import a new project into Git, make changes to it, and share changes with other developers.

If you are instead primarily interested in using Git to fetch a project, for example, to test the latest version, you may prefer to start with the first two chapters of The Git User's Manual.

It is a good idea to introduce yourself to Git with your name and public email address before doing any operation. The easiest way to do so is:

```
git config --global user.name "Your Name Comes Here"
git config --global user.email you@yourdomain.example.com
```

2.7.1 Importing a new project

Assume you have a tar-ball project.tar.gz with your initial work. You can place it under Git revision control as follows.

```
tar xzf project.tar.gz
cd project
git init
```

Git will reply with:

```
Initialized empty Git repository in .git/
```

You've now initialized the working directory–you may notice a new directory created, named ".git".

Next, tell Git to take a snapshot of the contents of all files under the current directory (note the .), with git add:

```
git add .
```

This snapshot is now stored in a temporary staging area which Git calls the "index". You can permanently store the contents of the index in the repository with git commit:

```
git commit
```

This will prompt you for a commit message. You've now stored the first version of your project in Git.

Making changes Modify some files, then add their updated contents to the index:

```
git add file1 file2 file3
```

You are now ready to commit. You can see what is about to be committed using git diff with the –cached option:

```
git diff --cached
```

(Without `-cached`, git diff will show you any changes that you've made but not yet added to the index.) You can also get a brief summary of the situation with git status:

```
git status
# On branch master
# Changes to be committed:
#   (use ``git reset HEAD <file>...'' to unstage)
#
#modified:   file1
#modified:   file2
#modified:   file3
#
```

If you need to make any further adjustments, do so now, and then add any newly modified content to the index. Finally, commit your changes with:

```
git commit
```

This will again prompt you for a message describing the change, and then record a new version of the project.

Alternatively, instead of running git add beforehand, you can use

```
git commit -a
```

which will automatically notice any modified (but not new) files, add them to the index, and commit, all in one step.

A note on commit messages: Though not required, it's a good idea to begin the commit message with a single short (less than 50 character) line summarizing the change, followed by a blank line and then a more thorough description. The text up to the first blank line in a commit message is treated as the commit title, and that title is used throughout Git. For example, git-format-patch(1) turns a commit into email, and it uses the title on the Subject line and the rest of the commit in the body.

Git tracks content not files Many revision control systems provide an add command that tells the system to start tracking changes to a new file. Git's add command does something simpler and more powerful: git add is used both for new and newly modified files, and in both cases it takes a snapshot of the given files and stages that content in the index, ready for inclusion in the next commit.

2.7.2 Viewing project history

At any point you can view the history of your changes using

```
git log
```

If you also want to see complete diffs at each step, use

```
git log -p
```

Often the overview of the change is useful to get a feel of each step

```
git log --stat --summary
```

Git is great for collaboration and allows creating and merging branches. When git projects are hosted on could services such as GitHub, you can send pull requests to other users which they can accept and reject. For more details consult the official documentation.

3
Python

Python is a general-purpose high-level programming language. Its design philosophy emphasizes programmer productivity and code readability. It has a minimalist core syntax with very few basic commands and simple semantics. It also has a large and comprehensive standard library, including an Application Programming Interface (API) to many of the underlying operating system (OS) functions. Python provides built-in objects such as linked lists (`list`), tuples (`tuple`), hash tables (`dict`), arbitrarily long integers (`long`), complex numbers, and arbitrary precision decimal numbers.

Python supports multiple programming paradigms, including object-oriented (`class`), imperative (`def`), and functional (`lambda`) programming. Python has a dynamic type system and automatic memory management using reference counting (similar to Perl, Ruby, and Scheme).

Python was first released by Guido van Rossum in 1991[**?**]. The language has an open, community-based development model managed by the non-profit Python Software Foundation. There are many interpreters and compilers that implement the Python language, including one in Java (Jython), one built on .Net (IronPython) and one built in Python itself (PyPy). In this brief review, we refer to the reference C implementation created by Guido.

You can find many tutorials, the official documentation, and library references of the language on the official Python website. [1]

For additional Python references, we can recommend the books in ref. [**?**] and ref. [**?**].

You may skip this chapter if you are already familiar with the Python language.

3.0.3 Python vs Java and C++ syntax

	Java/C++	**Python**
assignment	$a = b;$	$a = b$
comparison	if ($a == b$)	if $a == b$:
loops	for($a = 0; a < n; a++$)	for a in $range(0, n)$:
block	Braces {...}	indentation
function	$floatf$(float a) {	def $f(a)$:
function call	$f(a)$	$f(a)$
arrays/lists	$a[i]$	$a[i]$
member	a.member	a.member
nothing	$null\ /\ void*$	$None$

As in Java, variables which are primitive types (bool, int, float) are passed by copy but more complex types, unlike C++, are passed by reference.

3.0.4 help, dir

The Python language provides two commands to obtain documentation about objects defined in the current scope, whether the object is built-in or user-defined.

We can ask for `help` about an object, for example "1":

```
>>> help(1)
Help on int object:

class int(object)
 |  int(x[, base]) -> integer
 |
 |  Convert a string or number to an integer, if possible.  A floating point
 |  argument will be truncated towards zero (this does not include a string
 |  representation of a floating point number!)  When converting a string, use
 |  the optional base.  It is an error to supply a base when converting a
 |  non-string. If the argument is outside the integer range a long object
 |  will be returned instead.
 |
 |  Methods defined here:
 |
 |  __abs__(...)
 |      x.__abs__() <==> abs(x)
```

```
...
```

and, since "1" is an integer, we get a description about the `int` class and all its methods. Here the output has been truncated because it is very long and detailed.

Similarly, we can obtain a list of methods of the object "1" with the command `dir`:

```
>>> dir(1)
['__abs__', '__add__', '__and__', '__class__', '__cmp__', '__coerce__',
'__delattr__', '__div__', '__divmod__', '__doc__', '__float__',
'__floordiv__', '__getattribute__', '__getnewargs__', '__hash__', '__hex__',
'__index__', '__init__', '__int__', '__invert__', '__long__', '__lshift__',
'__mod__', '__mul__', '__neg__', '__new__', '__nonzero__', '__oct__',
'__or__', '__pos__', '__pow__', '__radd__', '__rand__', '__rdiv__',
'__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__',
'__rlshift__', '__rmod__', '__rmul__', '__ror__', '__rpow__', '__rrshift__',
'__rshift__', '__rsub__', '__rtruediv__', '__rxor__', '__setattr__',
'__str__', '__sub__', '__truediv__', '__xor__']
```

3.1 Python Types

Python is a dynamically typed language, meaning that variables do not have a type and therefore do not have to be declared. Variables may also change the type of value they hold through its life. Values, on the other hand, do have a type. You can query a variable for the type of value it contains:

```
>>> a = 3
>>> print(type(a))
<type 'int'>
>>> a = 3.14
>>> print(type(a))
<type 'float'>
>>> a = 'hello python'
>>> print(type(a))
<type 'str'>
```

Python also includes, natively, data structures such as lists and dictionaries.

3.1.1 `int` and `long`

There are two types representing integer numbers, they are `int` and `long`. The difference is `int` corresponds to the microprocessor's native bit length. Typically this is 32 bits and can hold signed integers in range $[-2^{31}, +2^{31})$ while the `long` type can hold almost any arbitrary integer. It is important that Python automatically converts one into the other as necessary and you can mix and match the two types in computations. Here is an example:

```
>>> a = 1024
>>> type(a)
<type 'int'>
>>> b = a**128
>>> print(b)
20815864389328798163850480654728171077230524494533409610638224700807216119346720
59602447888346464836968484322790856201558276713249664692981627981321135464152584
82590187784406915463666993231671009459188410953796224233873542950969577339250027
68876520583464697770622321657076833170056511209332449663781837603694136444406281
042053396870977465916057756101739472373801429441421111406337458176
>>> print(type(b))
<type 'long'>
```

Computers represent 32 bit integer numbers by converting them to base 2. The conversion works in the following way:

```
def int2binary(n, nbits=32):
    if n<0:
        return [1 if bit==0 else 0 for bit in int2binary(-n-1,nbits)]
    bits = [0]*nbits
    for i in range(nbits):
        n, bits[i] = divmod(n,2)
    if n: raise OverflowError
    return bits
```

The case $n < 0$ is called *two's complement* and is defined as the value obtained by subtracting the number from the largest power of two (2^{32} for 32 bits). Just by looking at the most significative bit one can determine the sign of the binary number (1 for negative and 0 for zero or positive).

3.1.2 `float` and `decimal`

There are two ways to represent decimal numbers in Python. Using the native double precision (64bits) representation, `float`, or using the `decimal` module.

Most numerical problems are dealt with simply using `float`:

```
>>> pi = 3.141592653589793
>>> two_pi = 2.0 * pi
```

The `cmath` module, described later, contains many math functions that use with the `float` type.

Floating point numbers are internally represented as follows:

$$x = \pm m 2^e \tag{3.1}$$

where x is the number, m is called mantissa and it is zero or a number in range [1,2), e is called exponent. The sign, m and e can be computed using the following algorithm which also writes their representation in binary:

```
def float2binary(x,nm=52,ne=11):
    if x==0:
        return 0, [0]*nm, [0]*ne
    sign,mantissa, exponent = (1 if x<0 else 0),abs(x),0
    while abs(mantissa)>=2:
        mantissa,exponent = 0.5*mantissa,exponent+1
    while 0<abs(mantissa)<1:
        mantissa,exponent = 2.0*mantissa,exponent-1
    return sign,int2binary(int(2**nm*(mantissa-1)),nm),int2binary(exponent,ne)
```

Because the mantissa is stored in a fixed number of bits (11 for a 64 bits floating point number) then exponents smaller than -1022 and larger than 1023 cannot be represented. An arithmetic operation that returns a number smaller than $2^{-1022} \simeq 10^{-308}$ cannot be represented and results in an Underflow error. An operation that returns a number larger than $2^{1023} \simeq 10^{308}$ also cannot be represented and results in an Overflow error.

Here is an example of Overflow:

```
>>> a = 10.0**200
>>> a*a
inf
```

And here is an example of Underflow:

```
>>> a = 10.0**-200
>>> a*a
0.0
```

Another problem with finite precision arithmetic is the loss of precision for computation. Consider the case of the difference between two numbers with very different orders of magnitude. In order to compute the difference the CPU reduces them to the same exponent (the largest of the two) and then computes the difference in the two mantissas. If two numbers differ for a factor 2^k than the mantissa of the smallest number, in binary, needs to be shifted by k position thus resulting in a loss of information because the k least significant bits in the mantissa are ignored. If the difference between the two numbers is greater than a factor 2^{52} all bits in the mantissa of the smallest number are ignored and the smallest number becomes completely invisible.

Below is a practical example that produces a wrong result:

```
>>> a = 1.0
>>> b = 2.0**53
>>> a+b-b
0.0
```

Simple example of what occurs internally in a processor to add two floating point numbers together. The IEEE 754 standard states for 32 bit floating point numbers the exponent has a range of -126 to +127.

```
262 in IEEE 754: 0 10000111 00000110000000000000000  (+ e:8 m:1.0234375)
  3 in IEEE 754: 0 10000000 10000000000000000000000  (+ e:1 m:1.5)
265 in IEEE 754: 0 10000111 00001001000000000000000
```

To add 262.0 to 3.0 the Exponents must be the same. The exponent of the lesser number is increased to the exponent of the greater number. In this case 3's exponent must be increased by 7. Increasing the exponent by 7 means the mantissa must be shifted 7 binary digits to the right.

```
0 10000111 00000110000000000000000
0 10000111 00000011000000000000000  (The implied '1' is also pushed 7 places to the
      right)
-----------------------------------
0 10000111 00001001000000000000000  which is the IEEE 754 format for 265.0
```

In the case of two numbers where the exponent is greater than the number of digits in the mantissa the smaller number is shifted right off the end. The effect is a zero is added to the larger number.

In some cases only some of the bits of the smaller number's mantissa are lost an a partial addition occurs.

This precision issue is always present but not always obvious. It may consist of a small discrepancy between the true value and the computed value. This difference may increase during the computation, in particular in iterative algorithms, and may be sizable in the result of a complex algorithm.

Python also has a module for decimal floating point arithmetic which allows decimal numbers to be represented exactly. The class Decimal incorporates a notion of significant places (unlike hardware based binary floating point, the decimal module has a user alterable precision):

```
>>> from decimal import Decimal, getcontext
>>> getcontext().prec = 28 # set precision
>>> Decimal(1) / Decimal(7)
Decimal('0.1428571428571428571428571429')
```

Decimal numbers can be used almost everywhere in place of floating point number arithmetics but are slower and should be used only where arbitrary precision arithmetics is required. It does not suffer from the overflow, underflow, and precision issues described above.

```
>>> from decimal import Decimal
>>> a = Decimal(10.0)**300
>>> a*a
Decimal('1.000000000000000000000000000E+600')
```

3.1.3 `str`

Python supports the use of two different types of strings: ASCII strings and Unicode strings. ASCII strings are delimited by `'...'`, `"..."`, `'''...'''`, or `"""..."""`. Triple quotes delimit multiline strings. Unicode strings start with a `u` followed by the string containing Unicode characters. A Unicode string can be converted into an ASCII string by choosing an encoding (for example UTF8):

```
>>> a = 'this is an ASCII string'
>>> b = u'This is a Unicode string'
>>> a = b.encode('utf8')
```

After executing these three commands, the resulting `a` is an ASCII string storing UTF8 encoded characters.

It is also possible to write variables into strings in various ways:

```
>>> print('number is ' + str(3))
number is 3
>>> print('number is %s' % (3))
number is 3
>>> print('number is %(number)s' % dict(number=3))
number is 3
```

The final notation is more explicit and less error prone, and is to be preferred.

Many Python objects, for example numbers, can be serialized into strings using `str` or `repr`. These two commands are very similar but produce slightly different output. For example:

```
>>> for i in [3, 'hello']:
...     print(str(i), repr(i))
3 3
hello 'hello'
```

For user-defined classes, `str` and `repr` can be defined/redefined using the special operators `__str__` and `__repr__`. These are briefly described later in this chapter. For more information on the topic, refer to the official Python documentation [**?**].

Another important characteristic of a Python string is that it is an iterable object, similar to a list:

```
>>> for i in 'hello':
...     print(i)
h
e
l
l
o
```

3.1.4 `list` and `array`

A list is a native mutable Python object made of a sequence of arbitrary objects. Arrays are not native to Python but unlike lists each element must be of the same type but are computationally faster.

The main methods of a Python lists and arrays are append, insert, and delete. Other useful methods include count, index, reverse and sort. Note: arrays do not have a sort method.

```
>>> b = [1, 2, 3]
>>> print(type(b))
<type 'list'>
>>> b.append(8)
>>> b.insert(2, 7) # insert 7 at index 2 (3rd element)
>>> del b[0]
>>> print(b)
[2, 7, 3, 8]
>>> print(len(b))
4
>>> b.append(3)
>>> b.reverse()
print(b," 3 appears ", b.count(3), " times.  The number 7 appears at index ", b.
     index(7))
[3, 8, 3, 7, 2] 3 appears 2 times.  The number 7 appears at index 3
```

Lists can be sliced:

```
>>> a= [2, 7, 3, 8]
>>> print(a[:3])
[2, 7, 3]
>>> print(a[1:])
[7, 3, 8]
>>> print(a[-2:])
[3, 8]
```

and concatenated/joined:

```
>>> a = [2, 7, 3, 8]
>>> a = [2, 3]
>>> b = [5, 6]
>>> print(a + b)
[2, 3, 5, 6]
```

A list is iterable; you can loop over it:

```
>>> a = [1, 2, 3]
>>> for i in a:
...     print(i)
1
2
3
```

There is a very common situation for which a *list comprehension* can be used. Consider the following code:

```
>>> a = [1,2,3,4,5]
>>> b = []
>>> for x in a:
...     if x % 2 == 0:
...         b.append(x * 3)
>>> print(b)
```

```
[6, 12]
```

This code clearly processes a list of items, selects and modifies a subset of the input list, and creates a new result list. This code can be entirely replaced with the following list comprehension:

```
>>> a = [1,2,3,4,5]
>>> b = [x * 3 for x in a if x % 2 == 0]
>>> print(b)
[6, 12]
```

Python has a module called `array`. It provides an efficient array implementation. Unlike lists, array elements must all be of the same type and the type must be either a char, short, int, long, float or double. A type of char, short, int or long may be either signed or unsigned.

```
>>> from array import array
>>> a = array('d',[1,2,3,4,5])
array('d',[1.0, 2.0, 3.0, 4.0, 5.0])
```

An array object can be used in the same way as a list but its elements must all be of the same type, specified by the first argument of the constructor ("d" for double, "l" for signed long, "f" for float, and "c" for character). For a complete list of avilable options, refer to the official Python documentation.

Using "array" over "list" can be faster but, more importantly, the "array" storage is more compact for large arrays.

3.1.5 `tuple`

A tuple is similar to a list, but its size and elements are immutable. If a tuple element is an object, the object itself is mutable but the reference to the object is fixed. A tuple is defined by elements separated by a comma and optionally delimited by round brackets:

```
>>> a = 1, 2, 3
>>> a = (1, 2, 3)
```

The round brakets are required for a tuple of zero elements such as

```
>>> a = () # this is an empty tuple
```

A trailing comma is required for a one element tuple but not for two or more elements.

```
>>> a = (1)   # not a tuple
>>> a = (1,)  # this is a tuple of one element
>>> b = (1,2) # this is a tuple of two elements
```

Since lists are mutable, this works:

```
>>> a = [1, 2, 3]
>>> a[1] = 5
>>> print(a)
[1, 5, 3]
```

the element assignment does not work for a tuple:

```
>>> a = (1, 2, 3)
>>> print(a[1])
2
>>> a[1] = 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

A tuple, like a list, is an iterable object. Notice that a tuple consisting of a single element must include a trailing comma, as shown below:

```
>>> a = (1)
>>> print(type(a))
<type 'int'>
>>> a = (1,)
>>> print(type(a))
<type 'tuple'>
```

Tuples are very useful for efficient packing of objects because of their immutability. The brackets are often optional. You may easily get each element of a tuple by assigning multiple variables to a tuple at one time:

```
>>> a = (2, 3, 'hello')
>>> (x, y, z) = a
>>> print(x)
2
>>> print(z)
hello
>>> a = 'alpha', 35, 'sigma' # notice the rounded brackets are optional
>>> p, r, q = a
print(r)
35
```

3.1.6 `dict`

A Python `dict`-ionary is a hash table that maps a key object to a value object. For example:

```
>>> a = {'k':'v', 'k2':3}
>>> print(a['k'])
v
>>> print(a['k2'])
3
>>> 'k' in a
True
>>> 'v' in a
False
```

You will notice that the format to define a dictionary is the same as JavaScript Object Notation [JSON]. Dictionaries may be nested:

```
>>> a = {'x':3, 'y':54, 'z':{'a':1,'b':2}}
>>> print(a['z'])
{'a': 1, 'b': 2}
>>> print(a['z']['a'])
1
```

Keys can be of any hashable type (int, string, or any object whose class implements the `__hash__` method). Values can be of any type. Different keys and values in the same dictionary do not have to be of the same type. If the keys are alphanumeric characters, a dictionary can also be declared with the alternative syntax:

```
>>> a = dict(k='v', h2=3)
>>> print(a['k'])
v
>>> print(a)
{'h2': 3, 'k': 'v'}
```

Useful methods are `has_key`, `keys`, `values`, `items` and `update`:

```
>>> a = dict(k='v', k2=3)
>>> print(a.keys())
['k2', 'k']
>>> print(a.values())
[3, 'v']
>>> a.update({'n1':'new item'})     # adding a new item
>>> a.update(dict(n2='newer item')) # alternate method to add a new item
>>> a['n3'] = 'newest item'         # another method to add a new item
>>> print(a.items())
```

```
[('k2', 3), ('k', 'v'), ('n3', 'newest item'), ('n2', 'newer item'), ('n1', 'new
    item')]
```

The `items` method produces a list of tuples, each containing a key and its associated value.

Dictionary elements and list elements can be deleted with the command `del`:

```
>>> a = [1, 2, 3]
>>> del a[1]
>>> print(a)
[1, 3]
>>> a = dict(k='v', h2=3)
>>> del a['h2']
>>> print(a)
{'k': 'v'}
```

Internally, Python uses the `hash` operator to convert objects into integers, and uses that integer to determine where to store the value. Using a key that is not hashable will cause an `unhashable type` error

```
>>> hash("hello world")
-1500746465
>>> k = [1,2,3]
>>> a = {k:'4'}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

3.1.7 `set`

A set is something between a list and dictionary. It represents a non-ordered list of unique elements. Elements in a set cannot be repeated. Internally it is implemented as a hash-table, similar to a set of keys in a dictionary. A set is created using the `set` constructor. Its argument can be a list, a tuple, or an iterator:

```
>>> s = set([1,2,3,4,5,5,5,5])  # notice duplicate elements are removed
>>> print(s)
set([1,2,3,4,5])
>>> s = set((1,2,3,4,5))
>>> print(s)
set([1,2,3,4,5])
>>> s = set(i for i in range(1,6))
>>> print(s)
set([1, 2, 3, 4, 5])
```

Since sets are non-ordered list, appending to the end is not applicable. Instead of `append`, add elements to a set using the `add` method:

```
>>> s = set()
>>> s.add(2)
>>> s.add(3)
>>> s.add(2)
>>> print(s)
set([2, 3])
```

Notice that the same element can not be added twice (2 in the example). The is no exception/error thrown when trying to add the same element more than once.

Since sets are non-ordered, the order you add items is not necessarily the order they will be returned.

```
>>> s = set([6,'b','beta',-3.4,'a',3,5.3])
>>> print (s)
set(['a', 3, 6, 5.3, 'beta', 'b', -3.4])
```

The `set` object supports normal set operations like union, intersection, and difference:

```
>>> a = set([1,2,3])
>>> b = set([2,3,4])
>>> c = set([2,3])
>>> print(a.union(b))
set([1, 2, 3, 4])
>>> print(a.intersection(b))
set([2, 3])
>>> print(a.difference(b))
set([1])
>>> if len(c) == len(a.intersection(c)):
...     print("c is a subset of a")
... else:
...     print("c is not a subset of a")
...
c is a subset of a
```

to check for membership:

```
>>> 2 in a
True
```

3.2 Python control flow statements

Python uses indentation to delimit blocks of code. A block starts with a line ending with colon and continues for all lines that have a similar or higher indentation as the next line. For example:

```
>>> i = 0
>>> while i < 3:
...     print(i)
...     i = i + 1
0
1
2
```

It is common to use four spaces for each level of indentation. It is a good policy not to mix tabs with spaces, which can result in (invisible) confusion.

3.2.1 `for...in`

In Python, you can loop over iterable objects:

```
>>> a = [0, 1, 'hello', 'python']
>>> for i in a:
...     print(i)
0
1
hello
python
```

In the example above you will notice that the loop index, 'i', takes on the values of each element in the list [0, 1, 'hello', 'python'] sequentially. Python `range` keyword creates a list of integers automatically, that may be used in a 'for' loop without manually creating a long list of numbers.

```
>>> a = range(0,5)
>>> print(a)
[0, 1, 2, 3, 4]
>>> for i in a:
...     print(i)
0
1
2
3
4
```

The parameters for `range(a,b,c)` are first parameter is the starting value of the list. The second parameter is next value if the list contained one more element. The third parameter is the increment value.

`range` can also be called with one parameter. It is matched to "b" above with the first paramater defaulting to 0 and the third to 1.

```
>>> print(range(5))
[0, 1, 2, 3, 4]
>>> print(range(53,57))
[53,54,55,56]
>>> print(range(102,200,10))
[102, 112, 122, 132, 142, 152, 162, 172, 182, 192]
>>> print(range(0,-10,-1))
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

`range` is very convenient for creating a list of numbers, however as the list grows in length, the memory required to store the list also grows. A more efficient option is to use the keyword is `xrange`. It generates an iterable range instead of generating the entire list of elements.

```
>>> for i in xrange(0, 4):
...     print(i)
0
1
2
3
```

This is equivalent to the C/C++/C#/Java syntax:

```
for(int i=0; i<4; i=i+1) { print(i); }
```

Another useful command is `enumerate`, which counts while looping and returns a tuple consisting of (index, value):

```
>>> a = [0, 1, 'hello', 'python']
>>> for (i, j) in enumerate(a):   # the ( ) around i, j are optional
...     print(i, j)
0 0
1 1
2 hello
3 python
```

There is also a keyword `range(a, b, c)` that returns a list of integers starting with the value `a`, incrementing by `c`, and ending with the last value smaller than `b`, `a` defaults to 0 and `c` defaults to 1. `xrange` is similar but does not actually generate the list, only an iterator over the list; thus it is better for looping.

You can jump out of a loop using `break`

```
>>> for i in [1, 2, 3]:
...     print(i)
...     break
1
```

You can jump to the next loop iteration without executing the entire code block with `continue`

```
>>> for i in [1, 2, 3]:
...     print(i)
...     continue
...     print('test')
1
2
3
```

Python also supports list comprehensions and you can build lists using using the following syntax:

```
>>> a = [i*i for i in [0, 1, 2, 3]:
>>> print(a)
[0, 1, 4, 9]
```

Sometimes you may need a counter to "count" the elements of a list while looping:

```
>>> a = [e*(i+1) for (i,e) in ['a','b','c','d']]
>>> print(a)
['a', 'bb', 'ccc', 'dddd']
```

3.2.2 `while`

Comparison operators in Python follow the C/C++/Java operators of ==, !=, ... etc. However Python also accepts the <> operator as not equal to and is equivalent to !=. Logical operators are `and`, `or` and `not`.

The `while` loop in Python works much as it does in many other programming languages, by looping an indefinite number of times and testing a condition before each iteration. If the condition is `False`, the loop ends.

```
>>> i = 0
>>> while i < 10:
...     i = i + 1
```

```
>>> print(i)
10
```

The `for` loop was introduced earlier in this chapter.

There is no `loop...until` or `do...while` construct in Python.

3.2.3 `if...elif...else`

The use of conditionals in Python is intuitive:

```
>>> for i in range(3):
...     if i == 0:
...         print('zero')
...     elif i == 1:
...         print('one')
...     else:
...         print('other')
zero
one
other
```

`elif` means "else if". Both `elif` and `else` clauses are optional. There can be more than one `elif` but only one `else` statement. Complex conditions can be created using the `not`, `and` and `or` logical operators.

```
>>> for i in range(3):
...     if i == 0 or (i == 1 and i + 1 == 2):
...         print('0 or 1')
```

3.2.4 `try...except...else...finally`

Python can throw - pardon, raise - Exceptions:

```
>>> try:
...     a = 1 / 0
... except Exception, e:
...     print('oops: %s' % e)
... else:
...     print('no problem here')
... finally:
...     print('done')
oops: integer division or modulo by zero
done
```

If an exception is raised, it is caught by the `except` clause and the `else` clause is not not executed. The `finally` clause is always executed.

There can be multiple `except` clauses for different possible exceptions:

```
>>> try:
...     raise SyntaxError
... except ValueError:
...     print('value error')
... except SyntaxError:
...     print('syntax error')
syntax error
```

The `finally` clause is guaranteed to be executed while the `except` and `else` are not. In the example below the function returns within a `try` block. This is bad practice, but it shows that the finally will execute regardless of the reason the `try` block is exited.

```
>>> def f(x):
...    try:
...        r = x*x
...        return r  # bad practice
...    except:
...        print("exception occurred %s" % e)
...    else:
...        print("nothing else to do")
...    finally:
...        print("Finally we get here")
...
>>> y = f(3)
Finally we get here
>>> print "result is ", y
result is  9
```

For every `try` you must have either a `except` or `finally` while the `else` is optional

Here is a list of built-in Python exceptions

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- Exception
      +-- GeneratorExit
      +-- StopIteration
      +-- StandardError
      |    +-- ArithmeticError
      |    |    +-- FloatingPointError
```

```
     |    |    +-- OverflowError
     |    |    +-- ZeroDivisionError
     |    +-- AssertionError
     |    +-- AttributeError
     |    +-- EnvironmentError
     |    |    +-- IOError
     |    |    +-- OSError
     |    |         +-- WindowsError (Windows)
     |    |         +-- VMSError (VMS)
     |    +-- EOFError
     |    +-- ImportError
     |    +-- LookupError
     |    |    +-- IndexError
     |    |    +-- KeyError
     |    +-- MemoryError
     |    +-- NameError
     |    |    +-- UnboundLocalError
     |    +-- ReferenceError
     |    +-- RuntimeError
     |    |    +-- NotImplementedError
     |    +-- SyntaxError
     |    |    +-- IndentationError
     |    |         +-- TabError
     |    +-- SystemError
     |    +-- TypeError
     |    +-- ValueError
     |    |    +-- UnicodeError
     |    |         +-- UnicodeDecodeError
     |    |         +-- UnicodeEncodeError
     |    |         +-- UnicodeTranslateError
     +-- Warning
          +-- DeprecationWarning
          +-- PendingDeprecationWarning
          +-- RuntimeWarning
          +-- SyntaxWarning
          +-- UserWarning
          +-- FutureWarning
          +-- ImportWarning
          +-- UnicodeWarning
```

For a detailed description of each of them, refer to the official Python documentation.

Any object can be raised as an exception, but it is good practice to raise objects that extend one of the built-in exception classes.

3.2.5 `def...return`

Functions are declared using `def`. Here is a typical Python function:

```
>>> def f(a, b):
...     return a + b
>>> print(f(4, 2))
6
```

There is no need (or way) to specify the type of an argument(s) or the return value(s). In this example, a function `f` is defined that can take two arguments.

Functions are the first code syntax feature described in this chapter to introduce the concept of *scope*, or *namespace*. In the above example, the identifiers `a` and `b` are undefined outside of the scope of function `f`:

```
>>> def f(a):
...     return a + 1
>>> print(f(1))
2
>>> print(a)
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    print(a)
NameError: name 'a' is not defined
```

Identifiers defined outside of function scope are accessible within the function; observe how the identifier `a` is handled in the following code:

```
>>> a = 1
>>> def f(b):
...     return a + b
>>> print(f(1))
2
>>> a = 2
>>> print(f(1) # new value of a is used)
3
>>> a = 1 # reset a
>>> def g(b):
...     a = 2 # creates a new local a
...     return a + b
>>> print(g(2))
4
>>> print(a # global a is unchanged)
1
```

If `a` is modified, subsequent function calls will use the new value of the global `a` because the function definition binds the storage location of the identifier a, not the value of a itself at the time of function declaration; however, if a is assigned-to inside function g, the global a is unaffected because the new local a hides the global value. The external-scope reference can be used in the creation of *closures*:

```
>>> def f(x):
...     def g(y):
...         return x * y
...     return g
>>> doubler = f(2) # doubler is a new function
>>> tripler = f(3) # tripler is a new function
>>> quadrupler = f(4) # quadrupler is a new function
>>> print(doubler(5))
10
>>> print(tripler(5))
15
>>> print(quadrupler(5))
20
```

Function f creates new functions; and note that the scope of the name g is entirely internal to f. Closures are extremely powerful.

Function arguments can have default values and can return multiple results as a tuple: (Notice the parentheses are optional and are omitted in the example.)

```
>>> def f(a, b=2):
...     return a + b, a - b
>>> x, y = f(5)
>>> print(x)
7
>>> print(y)
3
```

Function arguments can be passed explicitly by name, therefore the order of arguments specified in the caller can be different than the order of arguments with which the function was defined:

```
>>> def f(a, b=2):
...     return a + b, a - b
>>> x, y = f(b=5, a=2)
>>> print(x)
7
>>> print(y)
-3
```

Functions can also take a runtime-variable number of arguments. Parameters that start with * and ** must be the last two parameters. If the ** parameter is used it must be last in the list. Extra values passed in will be placed in the *identifier parameter while named values will be placed into the ft **identifier. Notice that when passing values into the function the unnamed values must be before any and all named values.

```
>>> def f(a, b, *extra, **extraNamed):
...     print "a = ", a
...     print "b = ", b
...     print "extra = ", extra
...     print "extranamed = ", extraNamed
>>> f(1, 2, 5, 6, x=3, y=2, z=6)
a =  1
b =  2
extra =  (5, 6)
extranamed =  {'y': 2, 'x': 3, 'z': 6}
```

Here the first two paramters (1 and 2) are matched with the paramters a and b while the tuple 5, 6 is placed into extra and the remaining items (which are in a dictionary format) are placed into extraNamed

In the opposite case, a list or tuple can be passed to a function that requires individual positional arguments by unpacking them:

```
>>> def f(a, b):
...     return a + b
>>> c = (1, 2)
>>> print(f(*c))
3
```

and a dictionary can be unpacked to deliver keyword arguments:

```
>>> def f(a, b):
...     return a + b
>>> c = {'a':1, 'b':2}
>>> print(f(**c))
3
```

3.2.6 lambda

lambda provides a way to define a short unnamed function:

```
>>> a = lambda b: b + 2
>>> print(a(3))
5
```

The expression "`lambda` [a]:[b]" literally reads as "a function with arguments [a] that returns [b]". The `lambda` expression is itself unnamed, but the function acquires a name by being assigned to identifier `a`. The scoping rules for `def` apply to `lambda` equally, and in fact the code above, with respect to `a`, is identical to the function declaration using `def`:

```
>>> def a(b):
...     return b + 2
>>> print(a(3))
5
```

The only benefit of `lambda` is brevity; however, brevity can be very convenient in certain situations. Consider a function called `map` that applies a function to all items in a list, creating a new list:

```
>>> a = [1, 7, 2, 5, 4, 8]
>>> map(lambda x: x + 2, a)
[3, 9, 4, 7, 6, 10]
```

This code would have doubled in size had `def` been used instead of `lambda`. The main drawback of `lambda` is that (in the Python implementation) the syntax allows only for a single expression; however, for longer functions, `def` can be used and the extra cost of providing a function name decreases as the length of the function grows. Just like `def`, `lambda` can be used to *curry* functions: new functions can be created by wrapping existing functions such that the new function carries a different set of arguments:

```
>>> def f(a, b): return a + b
>>> g = lambda a: f(a, 3)
>>> g(2)
5
```

Python functions, created with either `def` or `lambda` allow re-factoring of existing functions in terms of a different set of arguments.

3.3 Classes

Because Python is dynamically typed, Python classes and objects may seem odd. In fact, member variables (attributes) do not need to be specifically defined when declaring a class and different instances of the same class can have different attributes. Attributes are generally associated with the instance, not the class (except when declared as "class attributes", which is the same as "static member variables" in C++/Java).

Here is an example:

```
>>> class MyClass(object): pass
>>> myinstance = MyClass()
>>> myinstance.myvariable = 3
>>> print(myinstance.myvariable)
3
```

Notice that `pass` is a do-nothing command. In this case it is used to define a class `MyClass` that contains nothing. `MyClass()` calls the constructor of the class (in this case the default constructor) and returns an object, an instance of the class. The `(object)` in the class definition indicates that our class extends the built-in `object` class. This is not required, but it is good practice.

Here is a more involved class with multiple methods:

```
>>> class Complex(object):
...     z = 2
...     def __init__(self, real=0.0, imag=0.0):
...         self.real, self.imag = real, imag
...     def magnitude(self):
...         return (self.real**2 + self.imag**2)**0.5
...     def __add__(self,other):
...         return Complex(self.real+other.real,self.imag+other.imag)
>>> a = Complex(1,3)
>>> b = Complex(2,1)
>>> c = a + b
>>> print(c.magnitude())
5
```

Functions declared inside the class are methods. Some methods have special reserved names. For example, `__init__` is the constructor. In the example we created a class to store the `real` and the `imag` part of a complex number. The constructor takes these two variables and stores them into `self` (not a keyword but a variable that plays the same role as `this` in Java and `(*this)` in C++. (This syntax is necessary to avoid ambiguity when declaring nested classes, such as a class that is local to a method inside another class, something the Python allows but Java and C++ do not).

The `self` variable is defined by the first argument of each method. They all must have it but they can use another variable name. Even if we use another name, the first argument of a method always refers to the object calling the method. It plays the same role as the `this` keyword in the Java and the C++ languages.

`__add__` is also a special method (all special methods start and end in double underscore) and it overloads the `+` operator between `self` and `other`. In the example, `a+b` is equivalent to a call to `a.__add__(b)` and the `__add__` method receives `self=a` and `other=b`.

All variables are local variables of the method except variables declared outside methods which are called *class variables*, equivalent to C++ *static member variables* that hold the same value for all instances of the class.

3.3.1 Special methods and operator overloading

Class attributes, methods, and operators starting with a double underscore are usually intended to be private (*for example* to be used internally but not exposed outside the class) although this is a convention that is not enforced by the interpreter.

Some of them are reserved keywords and have a special meaning.

For example:

- `__len__`
- `__getitem__`
- `__setitem__`

They can be used, for example, to create a container object that acts like a list:

```
>>> class MyList(object):
>>>     def __init__(self, *a): self.a = list(a)
>>>     def __len__(self): return len(self.a)
>>>     def __getitem__(self, key): return self.a[key]
>>>     def __setitem__(self, key, value): self.a[key] = value
>>> b = MyList(3, 4, 5)
>>> print(b[1])
4
>>> b.a[1] = 7
>>> print(b.a)
[3, 7, 5]
```

Other special operators include `__getattr__` and `__setattr__`, which define the get and set methods (getters and setters) for the class, and `__add__`, `__sub__`, `__mul__`, `__div__` which overload arithmetic operators. For the use of these operators we refer the reader to the chapter on linear algebra where they will be used to implement algebra for matrices.

3.4 File input/output

In Python you can open and write in a file with:

```
>>> file = open('myfile.txt', 'w')
>>> file.write('hello world')
>>> file.close()
```

Similarly, you can read back from the file with:

```
>>> file = open('myfile.txt', 'r')
>>> print(file.read())
hello world
```

Alternatively, you can read in binary mode with "rb", write in binary mode with "wb", and open the file in append mode "a", using standard C notation.

The `read` command takes an optional argument, which is the number of bytes. You can also jump to any location in a file using `seek`.

You can read back from the file with `read`

```
>>> print(file.seek(6))
>>> print(file.read())
world
```

and you can close the file with:

```
>>> file.close()
```

In the standard distribution of Python, which is known as CPython, variables are reference-counted, including those holding file handles, so CPython knows that when the reference count of an open file handle decreases to zero, the file may be closed and the variable disposed. However, in other implementations of Python such as PyPy, garbage collection is used instead of reference counting, and this means that it is possible that there may accumulate too many open file handles at one time, resulting in an error before the *gc* has a chance to close and dispose of them all. Therefore it is best to explicitly close file handles when they are no longer needed.

3.5 `import` modules

The real power of Python is in its library modules. They provide a large and consistent set of Application Programming Interfaces (APIs) to many system libraries (often in a way independent of the operating system).

For example, if you need to use a random number generator, you can do:

```
>>> import random
>>> print(random.randint(0, 9))
5
```

This prints a random integer in the range of (0,9], 5 in the example. The function `randint` is defined in the module `random`. It is also possible to import an object from a module into the current namespace:

```
>>> from random import randint
>>> print(randint(0, 9))
```

or import all objects from a module into the current namespace:

```
>>> from random import *
>>> print(randint(0, 9))
```

or import everything in a newly defined namespace:

```
>>> import random as myrand
>>> print(myrand.randint(0, 9))
```

In the rest of this book, we will mainly use objects defined in modules `math`, `cmath`, `os`, `sys`, `datetime`, `time` and `cPickle`. We will also use the `random` module but we will describe it in a later chapter.

In the following subsections we consider those modules that are most useful.

3.5.1 `math` and `cmath`

Here is a sampling of some of the methods availble in the `math` and `cmath` packages.

- `math.isinf(x)` returns true if the floating point number ft x is positive or negative infinity
- `math.isnan(x)` returns true if the floating point number ft x is NaN. See Python documentation or IEEE 754 standards for more information.
- `math.exp(x)` returns ft e**x
- `math.log(x[, base]` returns the logarithm of ft x to the optional `base`. If ft base is not supplied ft e is assumed.
- `math.cos(x)`,`math.sin(x)`,`math.tan(x)` returns the cos, sin, tan of the value of `x`. `x` is in radians.
- `math.pi`, `math.e` the constants for ft pi and ft e to available precision

3.5.2 `os`

This module provides an interface to the operating system API. For example:

```
>>> import os
>>> os.chdir('..')
>>> os.unlink('filename_to_be_deleted')
```

Some of the `os` functions, such as `chdir`, are not thread-safe, *for example* they should not be used in a multi-threaded environment.

`os.path.join` is very useful; it allows the concatenation of paths in an OS-independent way:

```
>>> import os
>>> a = os.path.join('path', 'sub_path')
>>> print(a)
path/sub_path
```

System environment variables can be accessed via:

```
>>> print(os.environ)
```

which is a read-only dictionary.

3.5.3 `sys`

The `sys` module contains many variables and functions, but the used the most is `sys.path`. It contains a list of paths where Python searches for modules. When we try to import a module, Python searches the folders listed in `sys.path`. If you install additional modules in some location and want Python to find them, you need to append the path to that location to `sys.path`.

```
>>> import sys
>>> sys.path.append('path/to/my/modules')
```

3.5.4 `datetime`

The use of the datetime module is best illustrated by some examples:

```
>>> import datetime
>>> print(datetime.datetime.today())
2008-07-04 14:03:90
>>> print(datetime.date.today())
2008-07-04
```

Occasionally you may need to time-stamp data based on the UTC time as opposed to local time. In this case you can use the following function:

```
>>> import datetime
>>> print(datetime.datetime.utcnow())
2008-07-04 14:03:90
```

The `datetime` module contains various classes: `date`, `datetime`, `time` and `timedelta`. The difference between two date or two datetime or two time objects is a `timedelta`:

```
>>> a = datetime.datetime(2008, 1, 1, 20, 30)
>>> b = datetime.datetime(2008, 1, 2, 20, 30)
>>> c = b - a
>>> print(c.days)
1
```

We can also parse dates and datetimes from strings for example:

```
>>> s = '2011-12-31'
>>> a = datetime.datetime.strptime(s,'%Y-%m-%d')  #modified
>>> print(s.year, s.day, s.month)
2011 31 12  #modified
```

Notice that “%Y” matches the 4-digits year, “%m” matches the month as a number (1-12), “%d” matches the day (1-31), “%H” matches the hour, “%M” matches the minute, and “%S” matches the second. Check the Python documentation for more options.

3.5.5 `time`

The time module differs from `date` and `datetime` because it represents time as seconds from the epoch (beginning of 1970).

```
>>> import time
>>> t = time.time()
1215138737.571
```

Refer to the Python documentation for conversion functions between time in seconds and time as a `datetime`.

3.5.6 `urllib` and `json`

The `urllib` is a module to download data or a web page from a URL.

```
>>> page = urllib.urlopen('http://www.google.com/')
>>> html = page.read()
```

3.6 Regular Expressions

The most efficient way of searching for text/patterns in text is by using regular expressions. For example:

```
import urllib
import re
html = urllib.urlopen('http://www.depaul.com')
regex = re.compile('[\w\.]+@[\w\.]+')
emails = regex.findall(html)
print emails
```

`re.compile(...)` compiles the description of a patter into an object, `regex`. Th `findall` method of the object finds all expressions matching the compiled pattern. Here are useful rules:

- `^` means "beginning of text".
- `$` means "and of text of text".
- `.` means any character.
- `\w` means any alphanumeric character or underscore.
- `\W` means any character not alphanumeric character and not underscore.
- `\s` means any form of whitespace (space, tab, etc.).
- `\S` means any char but not a whitespace.
- `[...]` can be used to build an OR sequence of characters.
- `(...)` can be used to build a AND sequence of consecutive matching characters.
- `(...)*` means any repetition of the sequence `(...)`.
- `(...)+` means any non-zero repetition of the sequence `(...)`.

A slash can be used to escape special characters. In a string \ must be escaped as \\. An actual "\" must therefore be escaped twice \\\\.

Here are some examples of regular expressions (they are all approximations and oversimplifications):

- Match all email in a document

```
re.compile("[\w\.]+@[\w\.]").findall(html)
```

- Match all tags in HTML

```
re.compile("\<(\w+)").findall(html)
```

- Match all images sources in HTML

```
re.compile("""\<img [^>]*src=["'](.*)["']""").findall(html.lower())
```

- Match all links in HTML

```
re.compile("""\<a [^>]*href=["'](.*)["']""").findall(html.lower())
```

4 BeautifulSoup

Regular expressions are useful and indispensable, yet building the proper expressions to match generic tags, and tag attributes in HTML can be difficult. A better solution is to use a library designed specifically for this. One such library is BeautifulSoup. BeautifulSoup does not completely eliminate the need for regular expressions (in fact is uses regular expressions and you may have to build some to pass to some BeautifulSoup methods).

You can install BeautifulSoup from the Bash shell with

```
sudo easy_install BeautifulSoup
```

From within Python you only need to import one class `BeautifulSoup` which we shall rename `Soup` for brevity:

```
>>> from BeautifulSoup import BeautifulSoup as Soup
```

Given any help text, for example:

```
>>> html = "<html><p>Para 1<p>Para 2<blockquote>Quote 1<blockquote>Quote 2"
```

You can use the Soup object to parse it and construct an object representation of the Document Object Model:

```
>>> soup = Soup(html)
```

this object can be serialized back into HTML (although it would not produce the exact same HTML, but an equivalent one):

```
>>> print soup.prettify()
<html>
```

```
  <p>
   Para 1
  </p>
  <p>
   Para 2
   <blockquote>
    Quote 1
    <blockquote>
     Quote 2
    </blockquote>
   </blockquote>
  </p>
 </html>
```

The `soup` object (an instance of BeautifulSoup or BeautifulStoneSoup) is a deeply-nested, well-connected data structure that corresponds to the structure of an XML or HTML document. The parser object contains two other types of objects: Tag objects, which correspond to tags like the <TITLE> tag and the <B> tags; and NavigableString objects, which correspond to strings like “Page title” and “This is paragraph”.

The `soup` object can be used to query the DOM and manipulate it. For example, you can ask for the first `<p/>` tag which has an “align” attribute equal to “center”:

```
>>> p = soup.find('p', align='center')
```

The output `p` object is also a `BeautifulSoup` object. Unless you copy it, any change to this object will also affect the `soup` object that contains it.

Given then `p` object you can retrieve its attributes, for example its `id`, its `class`, its contents, or its serialized content:

```
>>> print p['id']
>>> print p['class']
>>> print p.contents
>>> print p.string
```

Similarly you can search all the tags in the `soup` object which match a given criteria, for example app `<p/>` tags:

```
>>> soup.findAll('p')
```

or all tags with an `id` that matches a regular expression:

```
>>> soup.findAll(id=re.compile("^x"))
```

An alternative syntax for getting all the `<p/>` tags is:

```
>>> tags = soup.p
```

The '.' notation can chained to obtain, for example the title inside the head of the document:

```
>>> print soup.head.title
```

You can also replace its contents with:

```
>>> soup.head.title.replaceWith('New Title')
```

Each `BeautifulSoup` object is a reference to a portion of the `soup` document. Any change will affect the original document.

Notice that any HTML document is a tree therefore you can traverse the tree by looking for the parent of a given node, its siblings (next and previous), as well as the next and previous tags in the DOM structure:

```
>>> print soup.head.parent.name
>>> print soup.p.nextSibling
>>> print soup.p.previousSibling
>>> print soup.p.next
>>> print soup.p.previous
```

5 Mechanize

You can install Mechanize from the Bash shell with

```
sudo easy_install mechanize
```

Mechanize simulates a web browser and its interaction with the web. You can start it with:

```
>>> import mechanize
>>> br = mechanize.Browser()
```

It is important to configure your browser so that it can fool third party web servers. Web servers set restriction and who can access web pages and in particular they do not like to be visited by robots. You have to pretend your browser is a "normal" one, for example Firefox:

```
br.set_handle_robots( False )
br.addheaders = [('User-agent', 'Firefox')]
```

Once you have a browser `br` you can give instructions to it, for example, "visit the Google web page":

```
>>> response = br.open("http://www.google.com/")
```

The response has attributes for example:

```
>>> print response.geturl()
http://www.google.com/
```

or the returned status code:

```
>>> print br.code
200
```

or the response headers

```
>>> print response.info().headers
['Date: Wed, 11 Sep 2013 18:09:10 GMT\r\n', 'Expires: -1\r\n', 'Cache-Control:
    private, max-age=0\r\n', 'Content-Type: text/html; charset=ISO-8859-1\r\n', '
    Set-Cookie: PREF=ID=a53b86f876c89fc9:U=548b6d330fff7040:FF=0:TM=1378922843:LM
    =1378922950:S=xPNYhTPc_HwsBqd8; expires=Fri, 11-Sep-2015 18:09:10 GMT; path=/;
     domain=.google.com\r\n', 'Server: gws\r\n', 'X-XSS-Protection: 1; mode=block\
    r\n', 'X-Frame-Options: SAMEORIGIN\r\n', 'Alternate-Protocol: 80:quic\r\n', '
    Connection: close\r\n']
```

Or a specific header:

```
>>> r.info().getheader('date')
'Wed, 11 Sep 2013 18:09:10 GMT'
```

Or the page content (HTML or binary depending on the data):

```
>>> print r.read()
```

What is most important is the ability of Mechanize to understand the page. You can ask it for example to follow a link and get another page. For example, follow a link that contains text matching a regular expressions:

```
>>> response = br.follow_link(text_regex=".*Google.*")
```

You can ask if the action was successful and the browser received HTML:

```
>>> assert br.viewing_html()
```

Because of Mechanize ability to understand HTML, you can ask it to locate all links in a page:

```
>>> for f in br.links(): print f
...
Link(base_url='http://www.google.com/', url='http://www.google.com/intl/en/options/
    ', text='More &#9660;', tag='a', attrs=[('class', 'gb3'), ('href', 'http://www
    .google.com/intl/en/options/'), ('onclick', 'this.blur();gbar.tg(event);return
     !1'), ('aria-haspopup', 'true')])
...
```

And/or all forms in the page:

```
>>> for f in br.forms(): print f
<f GET http://www.google.ca/search application/x-www-form-urlencoded
  <HiddenControl(ie=ISO-8859-1) (readonly)>
  <HiddenControl(hl=en) (readonly)>
  <HiddenControl(source=hp) (readonly)>
  <TextControl(q=)>
  <SubmitControl(btnG=Google Search) (readonly)>
  <SubmitControl(btnI=I'm Feeling Lucky) (readonly)>
  <HiddenControl(gbv=1) (readonly)>>
```

This says: There is only one form on the page/ Hidden field id's are ie (page encoding), hl (language code), hp (? don't know), and gbv (also don't know). The only not-hidden field id is q, which is a text input, which is the search text.

And you can ask it to select a form, fill it, and submit it:

```
>>> br.select_form( 'f' )
>>> br.form[ 'q' ] = 'foo'
>>> br.submit()
>>> response = br.response
```

6

Fabric

Fabric is a remote management tool written in Python.

You can install Fabric from the Bash shell with

```
sudo easy_install fabric
```

To use Fabric you create a fabfile (just a Python program) which defines the operations you want to perform for example:

```
from fabric.api import local

def dir():
  local('ls -l')
```

than you run it with:

```
fab -f /path/to/fabfile.py dir
```

This calls the uptime function inside the "fabfile.py". From now on we will assume the "fabfile.py" in the user root folder and will omit the `-f` command line option.

The `local(...)` function executes the shell command passed as argument. In this case the `ls -l` command. The command in the example is executed on the local machine but the point of Fabric is to execute the command remotely and simultaneously on one or more machines. Here is a better example

```
from fabric.api import env, run

env.hosts = [ '192.168.1.100', '192.168.1.101', '192.168.1.102' ]
env.user = "you"
env.password = "mysecret"
env.parallel = True

def dir():
    run('ls -l')
```

Now a call to

```
fab dir
```

will produce a directory listing from all the machines at the hosts defined in `env.hosts`. One can also specify a host using the `-H` command line option:

```
fab -H 192.168.1.100 dir
```

You can use `-h` to find out more about command line options:

```
Usage: fab [options] <command>[:arg1,arg2=val2,host=foo,hosts='h1;h2',...] ...

Options:
  -h, --help            show this help message and exit
  -d NAME, --display=NAME
                        print detailed info about command NAME
  -F FORMAT, --list-format=FORMAT
                        formats --list, choices: short, normal, nested
  -l, --list            print list of possible commands and exit
  --set=KEY=VALUE,...   comma separated KEY=VALUE pairs to set Fab env vars
  --shortlist           alias for -F short --list
  -V, --version         show program's version number and exit
  -a, --no_agent        don't use the running SSH agent
  -A, --forward-agent   forward local agent to remote end
  --abort-on-prompts    abort instead of prompting (for password, host, etc)
  -c PATH, --config=PATH
                        specify location of config file to use
  -D, --disable-known-hosts
                        do not load user known_hosts file
  -f PATH, --fabfile=PATH
                        python module file to import, e.g. '../other.py'
  --hide=LEVELS         comma-separated list of output levels to hide
  -H HOSTS, --hosts=HOSTS
                        comma-separated list of hosts to operate on
  -i PATH               path to SSH private key file. May be repeated.
  -k, --no-keys         don't load private key files from ~/.ssh/
  --keepalive=N         enables a keepalive every N seconds
  --linewise            print line-by-line instead of byte-by-byte
  -n M, --connection-attempts=M
                        make M attempts to connect before giving up
  --no-pty              do not use pseudo-terminal in run/sudo
  -p PASSWORD, --password=PASSWORD
```

```
                        password for use with authentication and/or sudo
  -P, --parallel        default to parallel execution method
  --port=PORT           SSH connection port
  -r, --reject-unknown-hosts
                        reject unknown hosts
  -R ROLES, --roles=ROLES
                        comma-separated list of roles to operate on
  -s SHELL, --shell=SHELL
                        specify a new shell, defaults to '/bin/bash -l -c'
  --show=LEVELS         comma-separated list of output levels to show
  --skip-bad-hosts      skip over hosts that can't be reached
  --ssh-config-path=PATH
                        Path to SSH config file
  -t N, --timeout=N     set connection timeout to N seconds
  -u USER, --user=USER  username to use when connecting to remote hosts
  -w, --warn-only       warn, instead of abort, when commands fail
  -x HOSTS, --exclude-hosts=HOSTS
                        comma-separated list of hosts to exclude
  -z INT, --pool-size=INT
                        number of concurrent processes to use in parallel mode
```

You can also declare and call functions which take arguments. For example here is a fabfile that allows you to apt-get install packages remotely:

```
from fabric.api import env, sudo

env.hosts = [ '192.168.1.100', '192.168.1.101', '192.168.1.102' ]
env.user = "you"
env.password = "mysecret"
env.parallel = True

def install(package):
   sudo('sudo apt-get -y install \%s' \% package)
```

and you can run it with:

```
fab install:python
```

A nice feature of Fabric is the ability to transfer files between the local and the remote host. Here is an example to copy a local file and unzip it at each at the remote destinations:

```
from fabric.api import env, sudo
from fabric.operations import put

env.hosts = [ '192.168.1.100', '192.168.1.101', '192.168.1.102' ]
env.user = 'you'
env.password = 'ysecret'
env.parallel = True
```

```

def deploy(filename):
    put(filename, '/home/you/destination/path')
    run('unzip /home/you/destination/path/\%s' \% filename)
```

which you can use with

```
fab deploy:myfile.zip
```

7
OpenStack

Bibliography

[1] http://www.python.org