CURRICULUM

INF3331 - fall 2013

Author: JOAKIM MYRVOLL

User: joakimmj

Due: 05.12.13
# Contents

## Bash

1. Basic bash scripting
   1.1 Variables
   1.2 If-statement
   1.3 Basic Calculator (bc)
   1.4 Catch signal
   1.5 Debugging
   1.6 Filetype
   1.7 File Writing
   1.8 Case
   1.9 File Reading
   1.10 for-loops
      1.10.1 C-style
   1.11 Bundle files
   1.12 Pipes
   1.13 Function
   1.14 Rename, copy and remove files
   1.15 Directory management
   1.16 Directory Tree Traversal
   1.17 Packing Directory Trees

## Python

2. Basic
   2.1 Running script
   2.2 Read Command Line Arguments
   2.3 Variables

3. Lists and Tuples
   3.1 List Functionality

4. Dictionary

5. String
   5.1 Operations

6. Loops
   6.1 While
   6.2 For
   6.3 Ranges

7. Function

8. Eval And Exec
   8.1 Namespaces

9. File reading and writing
# 10 Classes
- 10.1 Example ............................................. 14
- 10.2 Subclass ............................................ 15
- 10.3 Testing On The Class Type ..................... 15
- 10.4 Private/non-public data .......................... 15

# 11 Regular Expression
- 11.1 Search .............................................. 16
- 11.2 Find all ............................................ 17
- 11.3 Groups ............................................. 17
  - 11.3.1 Named Groups .............................. 17
- 11.4 Pattern-matching Modifiers .................... 17
- 11.5 Comments in a regex ............................ 18
- 11.6 Substitution ..................................... 18

# 12 Python scripting
- 12.1 Globbing .......................................... 18
- 12.2 Testing File Types ............................... 18
- 12.3 Copy, Rename and Remove Files ................ 19
- 12.4 Directory Management ........................... 19
- 12.5 Basename/directory of a path .................. 19
- 12.6 Traversing directory trees .................... 19
- 12.7 Tar Archives ..................................... 20
  - 12.7.1 Create ...................................... 20
  - 12.7.2 Read ......................................... 20
- 12.8 Argparse .......................................... 20
- 12.9 Persistence ....................................... 21
  - 12.9.1 Pickling .................................... 21
  - 12.9.2 Shelving .................................... 21
- 12.10 Running an application ....................... 22
  - 12.10.1 Running an application (old-style) .... 22
  - 12.10.2 Running applications and grabbing the output 22
  - 12.10.3 The new standard: subprocess ............ 22
- 12.11 Output pipe ..................................... 22

# 13 Simple Text Processing
- 13.1 Splitting text ................................... 23
- 13.2 Joining a list .................................... 23
- 13.3 Searching in strings ............................ 23
- 13.4 Substitution ..................................... 24

# 14 Numerical Python (NumPy)
- 14.1 Making Arrays ................................... 24
  - 14.1.1 Making float, int, complex arrays ....... 25
  - 14.1.2 Array with a sequence of numbers ....... 25
  - 14.1.3 Array construction from a Python list ... 26
  - 14.1.4 From anything to a NumPy array ........ 26
  - 14.1.5 Changing array dimensions ............... 26
  - 14.1.6 Array initialization from a Python function 27
  - 14.1.7 Array indexing ............................. 27
14.1.8 Other useful array operations ........................................ 27
14.2 Examples .............................................................. 27

15 Function .................................................................. 28
  15.1 Closure ............................................................... 28
  15.2 Decorators ........................................................... 28
  15.2.1 @decorator ......................................................... 29

16 Class ........................................................................ 29
  16.1 Static methods and class methods ................................. 29

17 Lambda ................................................................ 30

18 Map ........................................................................ 30

19 List-comprehension ...................................................... 30

20 Dictionary-comprehension ............................................. 30

21 Mixed language programming ........................................ 31
  21.1 Wrapper code .......................................................... 31
  21.2 SWIG .................................................................. 31
  21.3 Cython ................................................................. 31
  21.3.1 Example ............................................................. 32

22 Web applications ........................................................ 32
  22.1 CGI .................................................................... 32
  22.1.1 Debugging ........................................................ 33
  22.2 RESTful web services .............................................. 34
  22.3 Django ................................................................. 34

Index ........................................................................... 35
Part I

Bash

1 Basic bash scripting

Bourne shell (proprietary code)

```
#!/bin/sh
```

Bourne Again shell (open source)

```
#!/bin/bash
```

1.1 Variables

- Variables in Bash are untyped!
- Generally treated as character arrays, but permit simple arithmetic and other operations
- Variables can be explicitly declared to integer or array:
  ```
  declare -i i # i is an integer
  declare -a A # A is an array
  ```
- Assign a variable by `x = 3`, retrieve the value of the variable by `$x` (also called variable substitution).
- Variables passed as command line arguments when running a script are called positional parameters.
  ```
  $1 # first argument
  $2 # second argument
  # etc..
  ```

1.2 If-statement

```
if [ $i -eq 10 ]; then # integer comparison
  <body>
fi

if [ "$name" == "10" ]; then # string comparison
  <body>
fi
```

Unless you have declared a variable to be an integer, assume that all variables are strings and use double quotes (strings) when comparing variables in an if test

```
if [ "$?" != "0" ]; then # this is safe
  <body>
fi

if [ $? != 0 ]; then # might be unsafe
  <body>
fi
```
if [ "$option" == "-m" ]; then
  m=$1; shift; # load next command-line arg
elif [ "$option" == "-b" ]; then
  b=$1; shift;
else
  echo "$0: invalid option \"$option\"; exit
fi

1.3 Basic Calculator (bc)

#!/bin/sh
echo "Hello, World, $1 = $(echo $1 | bc -l)."

1.4 Catch signal

#!/bin/bash
# trap Ctrl-c
trap ctrl_c SIGINT

function ctrl_c () {
  echo "\*
  PROCESS TERMINATED
  *
  
  exit 0
}

while true
do
  echo "All work and no play makes Jack a dull boy."
  sleep 0.1
done

1.5 Debugging

Each source code line is printed prior to its execution if you add -x as option to /bin/sh or /bin/bash

Either in the header

#!/bin/sh -x

or on the command line:

unix> /bin/sh -x hw.sh
unix> sh -x hw.sh
unix> bash -x hw.sh

1.6 Filetype

if [ -d $dir ]; then # Directory?
  <body>
fi

if [ -f myfile ]; then # Plain file?
  <body>
```bash
if [ −x $myfile ]; then  # Executable?
  fi
fi
if [ −z $myfile ]; then  # Empty file?
  fi
if [ ! −z $myfile ]; then  # Not empty file?
  fi
```

## 1.7 File Writing

File writing is efficiently done by 'here documents':

```bash
cat > myfile <<EOF
multi-line text
and variable substitution such as
$myvariable is
supported. The final EOF must
start in column 1 of the
script file.
EOF
```

## 1.8 Case

```bash
case "$option" in
  −m) m=$1; shift ;; # load next command-line arg
  −b) b=$1; shift ;;
  *) echo "$0: invalid option "$option"; exit ;; esac
```

## 1.9 File Reading

```bash
cat myfile  # write myfile to the screen
cat myfile > yourfile  # write myfile to yourfile
cat myfile >> yourfile  # append myfile to yourfile
cat myfile | wc  # send myfile as input to wc
```

## 1.10 for-loops

```bash
for arg in $@;  # $@: <= all arguments
do
  <body>
done
for OUTPUT in $(Linux–Or–Unix–Command–Here)
```
1.10.1 C-style

```bash
declare -i i
for ((i=0; i<$n; i++)); do
  <body>
  done
```

1.11 Bundle files

```bash
#!/bin/sh
for i in $@; do
  echo "echo unpacking file $i"
  echo "cat > $i <<EOF"
  cat $i
  echo "EOF"
  done
```

Usage:

```bash
bundle file1 file2 > onefile # pack
bash onefile # unpack
```

Onofile:

```bash
echo unpacking file file1
cat > file1 <<EOF
<text from file1>
EOF
echo unpacking file file2
cat > file2 <<EOF
<text from file2>
EOF
```

1.12 Pipes

Output from one command can be sent as input to another command via a pipe

```bash
#!/bin/bash
function test {
  declare -i cnt
  for i in $@; do
    cnt = cnt + i
  done
```

1.13 Function
1.14 Rename, copy and remove files

```bash
# rename $myfile to tmp.1:
mv $myfile tmp.1
# force renaming:
mv -f $myfile tmp.1
# move a directory tree mytree to $root:
mv mytree $root
# copy myfile to $tmpfile:
cp myfile $tmpfile
# copy a directory tree mytree recursively to $root:
cp -r mytree $root
# remove myfile and all files with suffix .ps:
rm myfile *.ps
# remove a non-empty directory tmp/mydir:
rm -r tmp/mydir
```

1.15 Directory management

```bash
# make directory:
$dir = "mynewdir";
mkdir $dir
# move to $dir
cd $dir
# move to $HOME
cd
```

1.16 Directory Tree Traversal

Find all files larger than 2000 blocks a 512 bytes (=1Mb):

```bash
find $HOME -name "*" -type f -size +2000 -exec ls -s {} \;
```

Remove all these files:

```bash
find $HOME -name "*" -type f -size +2000 \ -exec ls -s {} \; -exec rm -f {} \;
```

or ask the user for permission to remove:

```bash
find $HOME -name "*" -type f -size +2000 \ -exec ls -s {} \; -ok rm -f {} \;
```

Find all files not being accessed for the last 90 days:

```bash
find $HOME -name "*" -atime +90 -print
```

and move these to /tmp/trash:

```bash
find $HOME -name "*" -atime +90 -print \ -exec mv -f {} /tmp/trash \;
```
### 1.17 Packing Directory Trees

The `tar` command can pack single files or all files in a directory tree into one file, which can be unpacked later.

```
$ tar -cvf myfiles.tar mytree file1 file2
# options:
# c: pack, v: list name of files, f: pack into file
# unpack the mytree tree and the files file1 and file2:
$ tar -xvf myfiles.tar
# options:
# x: extract (unpack)
```

The tarfile can be compressed:

```
$ gzip mytar.tar
# result: mytar.tar.gz
```

## Part II

**Python**

### 2 Basic

**Header:**

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

**Import libraries**

```
from math import sin  # import one module member
from math import *    # import everything from module
import math           # import module
import math as m      # import module and give it an alias
```

### 2.1 Running script

Run with command:

```
$ python test.py args
<print from script>
```

Linux alternative if file is executable (chmod 755 *.py):

```
$ ./test.py args
<print from script>
```

### 2.2 Read Command Line Arguments

```
import sys
x = float(sys.argv[1])  # sys.argv[0] is filename
```
2.3 Variables

Variables are not declared

Variables hold references to objects of any type

```python
1 a = 3  # reference to an int object containing 3
2 a = 3.0  # reference to a float object containing 3.0
3 a = '3.'  # reference to a string object containing '3.'
4 a = True  # reference to a boolean
5 a = ['1', 2]  # reference to a list object containing
6   # a string '1' and an integer 2
```

Test for a variable's type:

```python
1 if isinstance(a, int):  # int?
2 if isinstance(a, (list, tuple)):  # list or tuple?
```

3 Lists and Tuples

```python
1 mylist = ['a string', 2.5, 6, 'another string']
2 mytuple = ('a string', 2.5, 6, 'another string')
3 mylist[1] = -10
4 mylist.append('a third string')
5 mytuple[1] = -10  # illegal: cannot change a tuple
```

A tuple is a constant list (immutable)

3.1 List Functionality

```python
1 a = []  # initialize an empty list
2 a = [1, 4.4, 'run.py']  # initialize a list
3 a.append(elem)  # add 'elem' object to the end
4 a + [1, 3]  # add two lists
5 a[3]  # index a list element
6 a[-1]  # get last list element
7 a[1:3]  # slice: copy data to sublist (here: index 1, 2)
8 del a[3]  # delete an element (index 3)
9 a.remove(4.4)  # remove an element (with value 4.4)
10 a.index('run.py')  # find index corresponding to an element's value
11 'run.py' in a  # test if a value is contained in the list
12 a.count(v)  # count how many elements that have the value 'v'
13 len(a)  # number of elements in list 'a'
14 min(a)  # the smallest element in 'a'
15 max(a)  # the largest element in 'a'
16 sum(a)  # add all elements in 'a'
17 a.sort()  # sort list 'a' (changes 'a')
18 as = sorted(a)  # sort list 'a' (return new list)
19 a.reverse()  # reverse list 'a' (changes 'a')
20 b[3][0][2]  # nested list indexing
21 isinstance(a, list)  # is 'True' if 'a' is a list
```

4 Dictionary
5 String

Single- and double-quoted strings work in the same way:

```python
s1 = "some string with a number %g" % r
s2 = 'some string with a number %g' % r # = s1
```

Triple-quoted strings can be multi line with embedded newlines:

```python
text = """large portions of a text can be conveniently placed inside triple-quoted strings (newlines are preserved)"
```

Raw strings, where backslash is backslash:

```python
s3 = r'\ disobedient' 
```

With ordinary string (must quote backslash):

```python
s3 = '\ disobedient' 
```

5.1 Operations

```python
s = 'Berlin: 18.4 C at 4 pm'
s[8:17]  # extract substring
s.find(':')  # index where first ':' is found
s.split(':')  # split into substrings
s.split()  # split on whitespace
's.Berlin' in s  # test if substring is in s
s.replace('18.4', '20')  # lower case letters only
s.upper()  # upper case letters only
s.split()[4].isdigit()  # remove leading/trailing blanks
'. '.join(list_of_words)
```
6 Loops

6.1 While

```
while condition:
  <block of statements>
```

6.2 For

```
for element in somelist:
  <block of statements>
```

6.3 Ranges

```
range(start, stop, increment) # constructs an iterator.

# Typically, it is used in for-loops:
for i in range(10):
  print(i)
```

7 Function

```
def test(arg1, arg2, ..., argN, optional_arg='default value'):
  <body>
  return 'tada'

# function call:
print(test('hei', 23, ..., '8'))
```

8 Eval And Exec

Evaluating string expressions with eval:

```
>>> x = 20
>>> r = eval('x + 1.1')
>>> r
21.1
>>> type(r)
<type 'float'>
```

Executing strings with Python code, using exec:

```
exec(""
def f(x):
  return %s
""" % sys.argv[1])
```
8.1 Namespaces

exec and eval may take dictionaries for the global and local namespace:

```python
exec code in globals, locals
```

```python
eval(expr, globals, locals)
```

Example:

```python
a = 8; b = 9
d = { 'a':1, 'b':2} 
eval('a + b', d) # yields 3
```

and

```python
from math import *
d[ 'b' ] = pi 
eval('a+sin(b)', globals(), d) # yields 1
```

Creating such dictionaries can be handy

9 File reading and writing

```python
in file = open(filename, 'r')
```

```python
for line in in file:
    # process line
```

```python
lines = in file.readlines()
```

```python
for line in lines:
    # process line
```

```python
for i in xrange(len(lines)):
    # process lines[i] and perhaps next line lines[i+1]
```

```python
fstr = in file.read()
```

```python
# process the whole file as a string fstr
```

```python
in file.close()
```

```python
out file = open(filename, 'w') # new file or overwrite
```

```python
out file = open(filename, 'a') # append to existing file
```

```python
out file.write('""Some string
```

```python
""
```

10 Classes

```python
# String Representation Override Functions
```

```python
# Function Operator
```

```python
__str__ # str(A)
```

```python
__repr__ # repr(A)
```

```python
__unicode__ # unicode(x) (2.x only)
```

```python
# Attribute Override Functions
```

```python
# Function Indirect form Direct Form
```

```python
__getattr__ # getattr(A, B) A.B
```

Page 13 of 35
10.1 Example

class Test:
    
    Test
    
    def __init__(self, x, y, z):
        
        *** Constructing an instance ***
        
        self.x = x
        self.y = y
        self.z = z

    def __str__(self):
        
        *** Converting an object to a string ***
        
        return "(%g,%g,%g)" % (self.x, self.y, self.z)
def __repr__(self):
    """ Representation of the object in string form """
    return "Vec3D%(a) % str(self)

def __len__(self):
    """ Eucledian norm (length) """
    return int(math.sqrt(self.x**2+self.y**2+self.z**2))

def __getitem__(self, index):
    """ subscripting """
    if index==0: return self.x
    elif index==1: return self.y
    elif index==2: return self.z

def __setitem__(self, index, value):
    """ subscripting w/assignment """
    if index==0: self.x = value
    elif index==1: self.y = value
    elif index==2: self.z = value

All functions take self as first argument in the declaration, but not in the call

tmp = Test(6,9,8);

10.2 Subclass

Class SubTest is a subclass of Test:

class SubTest(Test):
    def __init__(self,x,y,z,k): # constructor
        Test.__init__(self,x,y,z)
        self.k = k;

10.3 Testing On The Class Type

Use isinstance for testing class type:

if isinstance(i2, MySub):
    # treat i2 as a MySub instance

Can test if a class is a subclass of another:

if issubclass(MySub, MyBase):
...

Can test if two objects are of the same class:

if inst1.__class__ is inst2.__class__

(is checks object identity, == checks for equal contents)

a.__class__ refers the class object of instance a

10.4 Private/non-public data
class MyClass:
def __init__(self):
    self._a = False  # non-public
    self.b = 0  # public
    self.__c = 0  # private

11 Regular Expression

\.
\^\$
\*
\+
?\[A-Z]\[^A-Z]\[abc\]\[^abc]\[^\^a-z\][A-Z]\[A-Z\]\[\ .\]*\[\^\[\[^no\]

\n\t\w\W\d\D\s\S\b\B\.\|\[\(\\\*

|  # any sequence of characters (except newline)
[.\^\]  # the characters . and ^
[^no]  # the string "no" at the beginning of a line
[^A-Z]  # the 3-character string 'A-Z' (A, minus, Z)
[A-Z]  # one of the chars A, B, C, ..., X, Y, or Z

11.1 Search

Finds first match

import re
class MyClass:
def __init__(self):
    self._a = False  # non-public
    self.b = 0  # public
    self.__c = 0  # private

11 Regular Expression

\.
\^\$
\*
\+
?\[A-Z]\[^A-Z]\[abc\]\[^abc]\[^\^a-z\][A-Z]\[A-Z\]\[\ .\]*\[\^\[\[^no\]

\n\t\w\W\d\D\s\S\b\B\.\|\[\(\\\*

|  # any sequence of characters (except newline)
[.\^\]  # the characters . and ^
[^no]  # the string "no" at the beginning of a line
[^A-Z]  # the 3-character string 'A-Z' (A, minus, Z)
[A-Z]  # one of the chars A, B, C, ..., X, Y, or Z

11.1 Search

Finds first match

import re
pattern = re.compile(r"t=(.*)\s{2}a:.+\s+(\d+)"
match = re.search(pattern, string_to_search)
if match:
    <do something>  # match.group(1), match.group(2)

11.2  Find all

Finds all matches

import re
pattern = re.compile(r"\b(%s)\b" % word_to_find, re.IGNORECASE)
all_matches = re.findall(pattern, string_to_search)

11.3  Groups

import re
pattern = r'\[(\d+):(\d+):?\s?((\d+))\]'  # 'from', 'to', 'step'
list_of_groups = re.search(pattern, string_interval).groups()

11.3.1  Named Groups

Use: (?P<name><regex>)

# Using named groups:
\\s*(?P<lower>\d+)\\s*(?P<upper>\d+)\\s*

# Extract groups by their names:
match.group('lower')
match.group('upper')

11.4  Pattern-matching Modifiers

if re.search('yes', answer, re.IGNORECASE):
    # pattern-matching modifier: re.IGNORECASE
    # now we get a match for 'yes', 'YES', 'Yes' ...
    # ignore case:
    re.I or re.IGNORECASE

    # let ^ and $ match at the beginning and
    # end of every line:
    re.M or re.MULTILINE

    # allow comments and white space:
    re.X or re.VERBOSE

    # let . (dot) match newline too:
    re.S or re.DOTALL

    # let e.g. \w match special chars (?, ?, ...):
    re.L or re.LOCALE
11.5 Comments in a regex

```python
# real number in scientific notation:
real_sn = r"\d+.d+ \[Ee\]?[+-]\d+d? # a number like 1.4098E-03, e-3, E+12"
match = re.search(real_sn, 'text with a=1.92E-04', re.VERBOSE)
```

11.6 Substitution

```python
# In general:
re.sub(pattern, replacement, str)

# Substitute float by double :
filestr contains a file as a string
filestr = re.sub('float', 'double', filestr)
```

12 Python scripting

12.1 Globbing

```python
# List all .ps and .gif files (Unix):
ls *.ps *,.gif

# Cross-platform way to do it in Python:
import glob
filelist = glob.glob('*.ps') + glob.glob('*.gif')
```

12.2 Testing File Types

```python
import os.path

if os.path.isfile(myfile): print('is a plain file')
if os.path.isdir(myfile): print('is a directory')
if os.path.islink(myfile): print('is a link')

# the size and age:
size = os.path.getsize(myfile)
time_of_last_access = os.path.getatime(myfile)
time_of_last_modification = os.path.getmtime(myfile)

# times are measured in seconds since 1970.01.01
days_since_last_access = (time.time() - os.path.getatime(myfile))/(3600*24)
```
12.3 Copy, Rename and Remove Files

```python
import shutil, os

# Copy a file:
shutil.copy(myfile, tmpfile)

# Rename a file:
os.rename(myfile, 'tmp.1')

# Remove a file:
os.remove('mydata') # or os.unlink('mydata')
```

12.4 Directory Management

```python
# Creating and moving to directories:
dirname = 'mynewdir'
if not os.path.isdir(dirname):
os.mkdir(dirname) # or os.mkdir(dirname, '0755')
os.chdir(dirname)

# Make complete directory path with intermediate directories:
p = os.path.join(os.environ['HOME'], 'py', 'src')
os.makedirs(p) # Unix: mkdirhier $HOME/py/src

# Remove a non-empty directory tree:
shutil.rmtree('myroot')
```

12.5 Basename/directory of a path

```python
# path
fname = '/home/hpl/scripting/python/intro/hw.py'

# basename: hw.py
basename = os.path.basename(fname)

# dirname: /home/hpl/scripting/python/intro
dirname = os.path.dirname(fname)

# or
dirname, basename = os.path.split(fname)

# extract suffix:
root, suffix = os.path.splitext(fname) # suffix: .py
```

12.6 Traversing directory trees

```python
root = os.environ['HOME'] # my home directory
os.path.walk(root, myfunc, arg)
```

arg is any user-defined argument, e.g. a nested list of variables.
Example:

```python
def do_something_with_files(arg, dirname, files):
    for file in files:
        # construct the file's complete path:
```
filename = os.path.join(dirname, file)
if os.path.isfile(filename):
    <do something with file>

root = os.environ['HOME']
os.path.walk(root, do_something_with_files, None)

12.7 Tar Archives
12.7.1 Create

import tarfile
files = 'NumPy_basics.py', 'hw.py', 'least squares.py'
tar = tarfile.open('tmp.tar.gz', 'w:gz') # gzip compression
for file in files:
    tar.add(file)

# check what's in this archive:
members = tar.getmembers() # list of TarInfo objects
for info in members:
    print('%s : size=%d, mode=%s, mtime=%s
          (info.name, info.size, info.mode,
           time.strftime('%Y.%m.%d', time.gmtime(info.mtime))))
tar.close()

Print:
NumPy_basics.py: size=11898, mode=33261, mtime=2004.11.23
hw.py: size=206, mode=33261, mtime=2005.08.12

12.7.2 Read

tar = tarfile.open('tmp.tar.gz', 'r')
for file in tar.getmembers():
    tar.extract(file) # extract file to current work.dir.

# do we have all the files?
allfiles = os.listdir(os.curdir)
for file in files:
    if not file in allfiles: print('missing', file)

hw = tar.extractfile('hw.py') # extract as file object
hw.readlines()

12.8 Argparse

import argparse

# Makes a parser for arguments
parser = argparse.ArgumentParser()
parser.add_argument('filename', help='file to search')
parser.add_argument('word', help='word to count')
12.9 Persistence

Many programs need to have persistent data structures, i.e., data live after the program is terminated and can be retrieved the next time the program is executed. `str`, `repr` and `eval` are convenient for making data structures persistent.

12.9.1 Pickling

Write any set of data structures to file using the cPickle module:

```
f = open(filename, 'w')
import pickle
pickle.dump(a1, f)
pickle.dump(a2, f)
pickle.dump(a3, f)
f.close()
```

Read data structures in again later:

```
f = open(filename, 'r')
a1 = pickle.load(f)
a2 = pickle.load(f)
a3 = pickle.load(f)
```

12.9.2 Shelving

Think of shelves as dictionaries with file storage

```
import shelve
database = shelve.open(filename)
database['a1'] = a1 # store a1 under the key 'a1'
database['a2'] = a2
database['a3'] = a3
# or
database['a123'] = (a1, a2, a3)

# retrieve data:
if 'a1' in database:
a1 = database['a1']
# and so on

# delete an entry:
del database['a2']
```

database.close()
12.10 Running an application

12.10.1 Running an application (old-style)

```python
# Run a stand-alone program:
cmd = 'myprog -c file1 -p -f -q > res'
failure = os.system(cmd)
if failure:
    print('%s: running myprog failed' % sys.argv[0])
sys.exit(1)

# Redirect output from the application to a list of lines:
pipe = os.popen(cmd)
output = pipe.readlines()
pipe.close()

for line in output:
    # process line
```

12.10.2 Running applications and grabbing the output

A nice way to execute another program:

```python
import commands
failure, output = commands.getstatusoutput(cmd)
if failure:
    print('Could not run', cmd; sys.exit(1))
for line in output.splitlines() # or output.split('
'):
    # process line

output holds both standard error and standard output (os.popen grabs only standard output so you do not see error messages). os.system, os.popen and the commands module are now replaced by the subprocess module
```

12.10.3 The new standard: subprocess

A module subprocess is the new standard for running stand-alone applications:

```python
from subprocess import call
try:
    returncode = call(cmd, shell=True)
    if returncode:
        print('Failure with returncode', returncode)
        sys.exit(1)
except OSError, message:
    print('Execution failed !n', message)
sys.exit(1)
```

More advanced use of subprocess applies its Popen object

```python
from subprocess import Popen, PIPE
p = Popen(cmd, shell=True, stdout=PIPE)
output, errors = p.communicate()
```

12.11 Output pipe

Open (in a script) a dialog with an interactive program:
```
pipe = Popen('gnuplot -persist', shell=True, stdin=PIPE).stdin
pipe.write('set xrange [0:10]; set yrange [-2:2]n')
pipe.write('plot sin(x)n')
pipe.write('quit') # quit Gnuplot
```

Same as "here documents" in Unix shells:
```
gnuplot <<EOF
set xrange [0:10]; set yrange [-2:2]
plot sin(x)
quit
EOF
```

## 13 Simple Text Processing

### 13.1 Splitting text

Split string into words:
```
files = 'case1.ps case2.ps case3.ps'
files.split() # whitespace = blank char, tab or newline
    # ['case1.ps', 'case2.ps', 'case3.ps']
```

Can split with other characters:
```
files = 'case1.ps, case2.ps, case3.ps'
files.split(',')
    # ['case1.ps', 'case2.ps', 'case3.ps']
files.split(',') # extra erroneous space after comma...
    # ['case1.ps, case2.ps, case3.ps'] -> unsuccessful split
```

### 13.2 Joining a list

Join is the opposite of split:
```
line1 = 'iteration 12: eps= 1.245E-05'
line1.split()
    # ['iteration', '12:', 'eps=', '1.245E-05']

w = line1.split()
    # Any delimiter text can be used:
    # iteration@@12@@eps=@@1.245E-05'

' '.join(w)
```

### 13.3 Searching in strings

Exact word match:
```
if line == "double":
    # line equals 'double'
```
```
if line.find('double') != -1:
    # line contains 'double'
```
Matching with Unix shell-style wildcard notation:

```python
import fnmatch

if fnmatch.fnmatch(line, 'double '):
    # line contains 'double'

# Here, double can be any valid wildcard expression, e.g., double* [Dd]ouble
```

Matching with full regular expressions:

```python
import re

if re.search('double ', line):
    # line contains 'double'

# Here, double can be any valid regular expression, e.g.,
# double[A-Za-z0-9_]* [Dd]ouble (DOUBLE| double)
```

### 13.4 Substitution

Simple substitution:

```python
newstring = oldstring.replace(substring, newsubstring)
```

Substitute regular expression `pattern` by replacement in `str`:

```python
import re
str = re.sub(pattern, replacement, str)
```

### 14 Numerical Python (NumPy)

- NumPy enables efficient numerical computing in Python
- NumPy is a package of modules, which offers efficient arrays (contiguous storage) with associated array operations coded in C or Fortran
- There are three implementations of Numerical Python
  - Numeric from the mid 90s (still widely used)
  - numarray from about 2000
  - numpy from 2006
- We use numpy (by Travis Oliphant)

```python
from numpy import *
```

#### 14.1 Making Arrays

```python
a = numpy.zeros(n)  # one-dim. array of length n
print(a) # n=4
# [ 0. 0. 0. 0.]

a = numpy.zeros((p,q,3)) # p=q=3 three-dim. array
print(a) # p=2, q=2
```
14.1.1 Making float, int, complex arrays

```python
a = numpy.zeros(3)
print(a.dtype)  # a's data type
# float64

a = numpy.zeros(3, int)
print(a)
# [0 0 0]
print(a.dtype)
# int32

a = numpy.zeros(3, float32)  # single precision
print(a)
# [0. 0. 0.]
print(a.dtype)
# float32

a = numpy.zeros(3, complex)
print(a)
# [0.+0.j, 0.+0.j, 0.+0.j]
print(a.dtype)
# complex128

# given an array a, make a new array of same dimension and data type:
x = numpy.zeros(a.shape, a.dtype)
```

14.1.2 Array with a sequence of numbers

`linspace(a, b, n)` generates n uniformly spaced coordinates, starting with a and ending with b

```python
x = numpy.linspace(-5, 5, 11)
print(x)
# [-5. -4. -3. -2. -1. 0. 1. 2. 3. 4. 5.]

# A special compact syntax is also available:
a = numpy.r_[-5:5:11j]  # same as linspace(-5, 5, 11)
print(a)
# [-5. -4. -3. -2. -1. 0. 1. 2. 3. 4. 5.]
```

`arange` works like `range(xrange)`

```python
x = numpy.arange(-5, 5, 1, float)
print(x)  # upper limit 5 is not included!!
# [-5. -4. -3. -2. -1. 0. 1. 2. 3. 4.]
```

Better to use a safer method: `seq(start, stop, increment)`

```python
from scitools.numpyutils import seq
x = seq(-5, 5, 1)
print(x)  # upper limit always included
# [-5. -4. -3. -2. -1. 0. 1. 2. 3. 4. 5.]
```
14.1.3 Array construction from a Python list

`array(list, [datatype])` generates an array from a list:

```python
pi = [0, 1.2, 4, -9.1, 5, 8]
a = array(pi)

# The array elements are of the simplest possible type:
z = array([1, 2, 3])
print(z) # array of integers
# [1 2 3]

z = array([1, 2, 3], float)
print(z)
# [ 1. 2. 3.]

# A two-dim. array from two one-dim. lists:
x = [0, 0.5, 1]; y = [-6.1, -2, 1.2] # Python lists
a = array([x, y]) # form array with x and y as rows

# From array to list:
alist = a.tolist()
```

14.1.4 From "anything" to a NumPy array

```python
# Given an object a
a = asarray(a) # converts a to a NumPy array (if possible/necessary)

# Arrays can be ordered as in C (default) or Fortran:
a = asarray(a, order='Fortran')
isfortran(a) # returns True if a's order is Fortran

# Use asarray to, e.g., allow flexible arguments in functions:
def myfunc(some_sequence):
a = asarray(some_sequence)
    return 3*a - 5

myfunc([1,2,3])  # list argument
myfunc((-1,1))   # tuple argument
myfunc(zeros(10)) # array argument
myfunc(-4.5)     # float argument
myfunc(6)        # int argument
```

14.1.5 Changing array dimensions

```python
a = array([[0, 1.2, 4, -9.1, 5, 8]])
a.shape = (2,3) # turn a into a 2x3 matrix
print(a)
# [[ 0. 1.2 4. ]
#  [-9.1 5. 8. ]]
print(a.size)
# 6

a.shape = (a.size,.) # turn a into a vector of length 6 again
print(a.shape)
# (6,)
prient(a)
# [ 0. 1.2 4. -9.1 5. 8. ]
```
14.1.6 Array initialization from a Python function

```python
def myfunc(i, j):
    return (i+1)*(j+4-i)

# make 3x6 array where a[i, j] = myfunc(i, j):
shape = (3, 6)
a = fromfunction(myfunc, shape)
pprint(a)
```

14.1.7 Array indexing

```python
# a is an array
a.clip(min=3, max=12)  # clip elements
a.mean()  # mean value
a.var()  # variance
a.std()  # standard deviation
median(a)
cov(x, y)  # covariance
trapezoid(a)  # Trapezoidal integration
diff(a)  # finite differences (da/dx)
```

14.2 Examples
dice1 = numpy.random.random_integers(1, 6, size=n)
numpy.where(dice1 == 6)

h = (b-a)/n
i = numpy.arange(1, n)
s = (f(a)+f(b))*(h/2) + h*sum(f(a+i*h))

# write out 2-column files with t and y[name] for each name:
for i in range(0, len(ynames)):
r =numpy.arange(0, len(tmp[:, i]))
numpy.savetxt(ynames[i]+'.dat', column_stack((r*dt, tmp[:, i][r])),
fmt='%12g %12.5e')

# the third line contains the name of the time series:
ynames = array(lines[2].split())

# Define a matrix and a vector
A = array([[1, 2, 3], [4, 5, 7], [6, 8, 10]], float)
b = array([-3, -2, -1], float)

do=numpy.array(do)
i=numpy.arange(0, len(do))
numpy.savetxt('report.txt', numpy.column_stack((do[i, 0], do[i, 1])), fmt='%15s %15s')

15 Function

15.1 Closure

def f():
    x = 3
    def inner():
        print "x = " , x
        return inner
    foo = f()
x = 10
foo()

# x = 3

15.2 Decorators

def f(x):
    return x**3 - 2

def checkrange(func):
    def inner(x):
        if x <0:
            print "out of range"
        else:
            return func(x)
    return inner
15.2.1 @decorator

```python
@checkrange
def g(x):
    return x**3-2

This is exactly the same as writing g=checkrange(g)
```

16 Class

16.1 Static methods and class methods

- New-style classes support static methods and class methods
- Both can be called without having an instance of the class
- Static method;
  - No knowledge of the class it belongs to
  - Declared as a regular function, without self or other class or instance related arguments
  - No implicit passing of instance or class when called
  - Defined using the decorator @staticmethod

```python
class A(object):
    @staticmethod
    def method1():
        pass
    # or old style; method1 = staticmethod(method1)
```

- Not widely used in Python

- Class method;
  - The first argument is the class, by convention named cls
  - When calling, the class is passed implicitly (just as with self for instance methods)
  - Defined using decorator @classmethod;
```python
class A(object):
    instances = {}

    @classmethod
    def method1(cls):
        print(cls.instances)
```

Commonly used as alternative constructors, to enable alternative ways of constructing an instance of the class

### 17 Lambda

\[\text{lambda <args>: <body>}\]

Example:

```python
f = lambda x: x**2
print(f(8))
# 64
```

### 18 Map

\[\text{map(function, iterable, ...)}\]

Example:

```python
l = list(map(lambda x, y: x**2+y, [2, 3, 4], [5,6]))
print(l)
# [9, 15]
```

### 19 List-comprehension

\[\{ f(x) \text{ for } x \text{ in iterable}\} \] # returns a list

### 20 Dictionary-comprehension

\[\{ \text{key: f(value) for } x \text{ in dictionary}\} \] # returns a dictionary
21 Mixed language programming

- Suppose we have a C function:
  
  ```c
  extern double hw1(double r1, double r2);
  ```

- We want to call this from Python as:
  
  ```python
  from hw import hw1
  r1 = 1.2; r2 = -1.2
  s = hw1(r1, r2)
  ```

- The Python variables `r1` and `r2` hold numbers (float), we need to extract these in the C code, convert to double variables, then call `hw1`, and finally convert the double result to a Python float

- All this conversion is done in wrapper code

21.1 Wrapper code

- Every object in Python is represented by C struct `PyObject`

- Wrapper code converts between `PyObject` variables and plain C variables (from `PyObject r1` and `r2` to double, and double result to `PyObject`):
  
  ```c
  static PyObject * _wrap_hw1 (PyObject *self, PyObject *args) {
    PyObject *resultobj;
    double arg1, arg2, result;
    PyArg_ParseTuple (args, (char *)"dd:hw1", &arg1, &arg2);
    result = hw1(arg1, arg2);
    resultobj = PyFloat_FromDouble(result);
    return resultobj;
  }
  ```

21.2 SWIG

A wrapper function is needed for each C function we want to call from Python. Wrapper codes are tedious to write. There are tools for automating wrapper code development. We shall use SWIG (for C/C++)

21.3 Cython

- Cython is a superset of Python, with additional functionality for defining C types and calling C functions

- Cython generates C wrapper code, which is compiled into a Python extension module

- Major advantage: enables incremental code optimization

- `cdef` is used to declare C variables:
  
  ```c
  cdef int i, j, k
  cdef float f, g[42], *h
  ```
Function arguments and return types may be declared:

```python
def foo(int i, char *s)
cdef int eggs(int i, float f):
cpdef double foo_2(int i float f):
```

If no type is specified for a variable, parameter or return type, it defaults to a Python object.

The standard Python for-loop is used in Cython:

```python
for i in range(n):
...
```

If `i` is declared as an integer (with `cdef int i`), this will be optimized into a standard C loop.

### 21.3.1 Example

**Python**

```python
from math import sin
def f(x):
    return sin(x ** 2)
def integrate_f(a, b, N):
s=0
dx = (b−a)/N
for i in xrange(N):
s += f(a+i*dx)
return s*dx
```

Takes around 3.5 seconds with N=1000000

**Cython**

```python
cdef extern from "math.h":
double sin(double arg)
cdef double f(double x):
    return sin(x**2)
cpdef double integrate_f(double a, double b, int N):
cdef double s=0
cdef double dx = (b−a)/N
cdef int i
for i in range(N):
s += f(a+i*dx)
return s*dx
```

A fully typed version runs about 10 times faster.
Speedup can be much higher, but requires slightly more complex example (loops within loops . . . )

### 22 Web applications

#### 22.1 CGI

Here: text ('Hello, World!'), text entry (for `r`) and a button 'equals' for computing the sine of `r`

**HTML code**
22.1.1 Debugging

- What happens if the CGI script contains an error?
  - Browser just responds "Internal Server Error" – a nightmare
- Start your Python CGI scripts with
  
  ```python
  import cgitb; cgitb.enable()
  ```
  
  to turn on nice debugging facilities:
  
  - Python errors now appear nicely formatted in the browser
22.2 RESTful web services

22.3 Django
Index

Bash, 4
  bc, 5
  Bundle, 7
  Case, 6
  Copy file, 8
  Debugging, 5
  Directory management, 8
  Filetype, 5
  Find, 8
  For-loops, 6
  Function, 7
  If-statement, 4
  Pipe, 7
  Read file, 6
  Remove file, 8
  Rename file, 8
  Tar and gzip, 9
  Variables, 4
  While-loops, 5
  Write file, 6

Python, 9
  Basename/directory of a path, 19
  Class, 13, 29
  Copy, rename and remove files, 19
  Dictionary, 10
  Dictionary-comprehension, 30
  Directory management, 19
  Eval, 12
  Exec, 12
  Execute, 9
  Function, 12, 28
  Globbing, 18
  Join, 23
  Lambda, 30
  List, 10
  List-comprehension, 30
  Loops, 12
    For, 12
    While, 12
  Map, 30
  Mixed language programming, 31
    Cython, 31
    SWIG, 31
  Numerical (NumPy), 24
    Array, 24

  Output pipe, 22
  Parsing command-line arguments, 20
  Pickling, 21
  Reading from file, 13
  Regex, 16, see Regular Expression
  Running an application, 22
  Scitools, 25
  Scripting, 18
  Search, 23
  Shelving, 21
  Split, 23
  String, 11
  Substitution, 24
  Tar archives, 20
  Testing file types, 18
  Text processing, 23
  Traversing directory trees, 19
  Tuple, 10
  Variables, 10
  Web applications, 32
    CGI, 32
    Django, 34
    RESTful web services, 34
  Writing to file, 13

  Regular Expression, 16
    Comments, 18
    Find all, 17
    Groups, 17
    Named, 17
    Search, 16
    Substitution, 18

Table Of Content, 3