CURRICULUM

INF3331 - fall 2013

Author:
JOAKIM MYRVOLL

User:
joakimmj

Due: 05.12.13

Contents

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 $x = 3.4$, 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 an 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>
```

```
fi

if [ -x $myfile ]; then    # Executable?
  <body>
fi

if [ -z $myfile ]; then    # Empty file?
  <body>
fi

if [ ! -z $myfile ]; then    # Not empty file?
  <body>
fi
```

1.7 File Writing

File writing is efficiently done by 'here documents':

```
cat > myfile <<EOF
multi-line text
can now be inserted here,
and variable substition such as
$myvariable is
supported. The final EOF must
start in column 1 of the
script file.
EOF
```

1.8 Case

```
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

```
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

```
for arg in $@;         # $@; <= all arguments
do
  <body>
done

for OUTPUT in $(Linux-Or-Unix-Command-Here)
```

```
do
  <body>
done
```

1.10.1 C-style

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

1.11 Bundle files

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

Usage:

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

Onefile:

```
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

```
# send files with size to sort -rn
# (reverse numerical sort) to get a list
# of files sorted after their sizes:
/bin/ls -s | sort -r
```

1.13 Function

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

```
    echo cnt
}

res='test 1.2 6 -998.1 1 0.1'
```

1.14 Rename, copy and remove files

```
# rename $myfile to tmp.1:
mv $myfile tmp.1
# force renaming:
mv -f $myfile tmp.1
# move a directory tree my tree 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

```
# 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):

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

Remove all these files:

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

or ask the user for permission to remove:

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

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

```
find $HOME -name '*' -atime +90 -print
```

and move these to /tmp/trash:

```
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 Aruments

```
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

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

Test for a variable's type:

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

3 Lists and Tuples

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

A tuple is a constant list (immutable)

3.1 List Functionality

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

4 Dictionary

```
a = {}                          # initialize an empty dictionary
a = {'point':[2,7], 'value':3}  # initialize a dictionary
a = dict(point=[2,7], value=3)  # initialize a dictionary
a['hide'] = True                # add new key-value pair to a dictionary
a['point']                      # get value corresponding to key point
'value' in a                    # 'True' if 'value' is a key in the dictionary
del a['point']                  # delete a key-value pair from the dictionary
a.keys()                        # list of keys
a.values()                      # list of values
len(a)                          # number of key-value pairs in dictionary 'a'
for key in a:                   # loop over keys in unknown order
for key in sorted(a.keys()):    # loop over keys in alphabetic order
isinstance(a, dict)             # is 'True' if 'a' is a dictionary
```

5 String

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

```
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:

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

Raw strings, where backslash is backslash:

```
s3 = r'\(\s+\.\d+\)'
```

With ordinary string (must quote backslash):

```
s3 = '\\(\\s+\\.\\d+\\)'
```

5.1 Operations

```
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
'Berlin' in s                   # test if substring is in s
s.replace('18.4', '20')
s.lower()                       # lower case letters only
s.upper()                       # upper case letters only
s.split()[4].isdigit()
s.strip()                       # 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:

```
exec code in globals, locals
eval(expr, globals, locals)
```

Example:

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

and

```
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

```
infile = open(filename, 'r')

for line in infile:
  # process line

lines = infile.readlines()
for line in lines:
# process line

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

fstr = infile.read()
# process the whole file as a string fstr

infile.close()
```

```
outfile = open(filename, 'w') # new file or overwrite
outfile = open(filename, 'a') # append to existing file
outfile.write("""Some string
  ....
  """)
```

10 Classes

```
# String Representation Override Functions
# Function      Operator
__str__         # str(A)
__repr__        # repr(A)
__unicode__     # unicode(x) (2.x only)

# Attribute Override Functions
# Function      Indirect form          Direct Form
__getattr__     # getattr(A, B)        A.B
```

```
  __setattr__     # setattr(A, B, C)      A.B = C
  __delattr__     # delattr(A, B)         del A.B

# Binary Operator Override Functions
# Function      Operator
  __add__         # A + B
  __sub__         # A - B
  __mul__         # A * B
  __truediv__     # A / B
  __floordiv__    # A // B
  __mod__         # A % B
  __pow__         # A ** B
  __and__         # A & B
  __or__          # A | B
  __xor__         # A ^ B
  __eq__          # A == B
  __ne__          # A != B
  __gt__          # A > B
  __lt__          # A < B
  __ge__          # A >= B
  __le__          # A <= B
  __lshift__      # A << B
  __rshift__      # A >> B
  __contains__    # A in B
# A not in B
# r in front of operater gives:
  __rtruediv__    # B / A

# Unary Operator Override Functions
# Function      Operator
  __pos__         # +A
  __neg__         # -A
  __inv__         # ~A
  __abs__         # abs(A)
  __len__         # len(A)

# Item Operator Override Functions
# Function      Operator
  __getitem__     # C[i]
  __setitem__     # C[i] = v
  __delitem__     # del C[i]
  __getslice__    # C[s:e]
  __setslice__    # C[s:e] = v
  __delslice__    # del C[s:e]
```

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)
```

```
14
15      def __repr__(self):
16          """ Representation of the object in string form """
17          return "Vec3D%s" % str(self)
18
19      def __len__(self):
20          """ Eucledian norm (length) """
21          return int(math.sqrt(self.x**2+self.y**2+self.z**2))
22
23      def __getitem__(self,index):
24          """ subscripting """
25          if index==0: return self.x
26          elif index==1: return self.y
27          elif index==2: return self.z
28
29      def __setitem__(self,index,value):
30          """ subscripting w/assignment """
31          if index==0: self.x = value
32          elif index==1: self.y = value
33          elif index==2: self.z = value
```

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

```
1  tmp = Test(6,9,8);
```

10.2 Subclass

Class SubTest is a subclass of Test:

```
1  class SubTest(Test):
2  def __init__(self,x,y,z,k): # constructor
3    Test.__init__(self,x,y,z)
4    self.k = k;
```

10.3 Testing On The Class Type

Use *isinstance* for testing class type:

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

Can test if a class is a subclass of another:

```
1  if issubclass(MySub, MyBase):
2    ...
```

Can test if two objects are of the same class:

```
1  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

```
.       # any single character except a newline
^       # the beginning of the line or string
$       # the end of the line or string
*       # zero or more of the last character
+       # one or more of the last character
?       # zero or one of the last character
[A-Z]   # matches all upper case letters
[abc]   # matches either a or b or c
[^b]    # does not match b
[^a-z]  # does not match lower case letters

.*      # any sequence of characters (except newline)
[.*]    # the characters . and
^no     # the string 'no' at the beginning of a line
[^no]   # neither n nor o
A-Z     # the 3-character string 'A-Z' (A, minus, Z)
[A-Z]   # one of the chars A, B, C, ..., X, Y, or Z

\n      # a newline
\t      # a tab
\w      # any alphanumeric (word) character the same as [a-zA-Z0-9_]
\W      # any non-word character the same as [^a-zA-Z0-9_]
\d      # any digit, same as [0-9]
\D      # any non-digit, same as [^0-9]
\s      # any whitespace character: space, tab, newline, etc
\S      # any non-whitespace character
\b      # a word boundary, outside [] only
\B      # no word boundary

\.      # a dot
\|      # vertical bar
\[      # an open square bracket
\)      # a closing parenthesis
\*      # an asterisk
\^      # a hat
\/      # a slash
\\      # a backslash
\{      # a curly brace
\?      # a question mark

|       # or operator (e.g. (eg|le)gs -- matches eggs or legs)
```

11.1 Search

Finds first match

```
import re

pattern = r"t=(.*)\s{2}a:.*\s+(\d+)"
```

```
4
5   match = re.search(pattern, string_to_search)
6   if match:
7     <do something> # match.group(1), match.group(2)
```

11.2 Find all

Finds all matches

```
1   import re
2
3   pattern = re.compile(r"\b(%s)\b" % word_to_find, re.IGNORECASE)
4   all_matches = re.findall(pattern, string_to_search)
```

11.3 Groups

```
1   import re
2
3   pattern = r'\[(\d+):(\d+),?\s?(\d*)\]'  # ('from', 'to', 'step')
4   list_of_groups = re.search(pattern, string_interval).groups()
```

11.3.1 Named Groups

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

```
1   # Using named groups:
2   \[\s*(?P<lower>-?\d+)\s*,\s*(?P<upper>-?\d+)\s*\]
3
4   # Extract groups by their names:
5   match.group('lower')
6   match.group('upper')
```

11.4 Pattern-matching Modifiers

```
1   if re.search('yes', answer, re.IGNORECASE):
2   # pattern-matching modifier: re.IGNORECASE
3   # now we get a match for 'yes', 'YES', 'Yes' ...
4   # ignore case:
5   re.I or re.IGNORECASE
6
7   # let ^ and $ match at the beginning and
8   # end of every line:
9   re.M or re.MULTILINE
10
11  # allow comments and white space:
12  re.X or re.VERBOSE
13
14  # let . (dot) match newline too:
15  re.S or re.DOTALL
16
17  # let e.g. \w match special chars (?, ?, ...):
18  re.L or re.LOCALE
```

11.5 Comments in a regex

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

# or when using compile:
c = re.compile(real_sn, re.VERBOSE)
match = c.search('text with a=1.9672E-04 ')
```

11.6 Substitution

```
# 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

```
# 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')
# referred to as file globbing
```

12.2 Testing File Types

```
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

```
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

```
# 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:
path = os.path.join(os.environ['HOME'],'py','src')
os.makedirs(path) # Unix: mkdirhier $HOME/py/src

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

12.5 Basename/directory of a path

```
# 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

```
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:

```
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', 'leastsquares.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
leastsquares.py: size=1560, mode=33261, mtime=2004.09.
```

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")
```

```
  parser.add_argument('-i', help="case insensitive count",action="store_true")
  parser.add_argument('-b', help="respect word boundaries",action="store_true")
  args = parser.parse_args()

  # Open file to read
  infile = open(args.filename, 'r')

  # Pattern to search
  pattern = args.word
```

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)

```
# Run a stand-alone program:
cmd = 'myprog -c file.1 -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:

```
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('\n'):
  # process line
# (output holds the output as a string)
```

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-al one applications:

```
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

```
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()
' '.join(w) # join w elements with delimiter ' '
# 'iteration 12: eps= 1.245E-05'

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

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:

```
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:

```
import re
if re.search(r'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:

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

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

```
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)

```
from numpy import *
```

14.1 Making Arrays

```
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
```

```
# [[[ 0. 0. 0.]
#   [ 0. 0. 0.]]
#  [[ 0. 0. 0.]
#   [ 0. 0. 0.]]]

a.shape                 # a's dimension
# (2, 2, 3)
```

14.1.1 Making float, int, complex arrays

```
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

```
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)

```
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)*

```
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:

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

# 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

```
# 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

```
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,)
print(a)
# [ 0. 1.2 4. -9.1 5. 8. ]
```

```
14
15  a = a.reshape(2,3) # same effect as setting a.shape
16  print(a.shape)
17  # (2, 3)
```

14.1.6 Array initialization from a Python function

```
1   def myfunc(i, j):
2     return (i+1)*(j+4-i)
3
4   # make 3x6 array where a[i,j] = myfunc(i,j):
5   a = fromfunction(myfunc, (3,6))
6   print(a)
7   # [[ 4., 5., 6., 7., 8., 9.],
8   #  [ 6., 8., 10., 12., 14., 16.],
9   #  [ 6., 9., 12., 15., 18., 21.]]
```

14.1.7 Array indexing

```
1   a = linspace(-1, 1, 6)
2   a[2:4] = -1      # set a[2] and a[3] equal to -1
3   a[-1] = a[0]     # set last element equal to first one
4   a[:] = 0         # set all elements of a equal to 0
5   a.fill(0)        # set all elements of a equal to 0
6   a.shape = (2,3) # turn a into a 2x3 matrix
7   print(a[0,1])    # print element (0,1)
8   a[i,j] = 10      # assignment to element (i,j)
9   a[i][j] = 10     # equivalent syntax (slower)
10  print(a[:,k])    # print column with index k
11  print(a[1,:])    # print second row
12  a[:,:] = 0       # set all elements of a equal to 0
13  a[::3,2::2]      # a[i,j] for i=0,3 and j=2,4
14  a[1:3,::2]       # a[i,j] for i=1,2 and j=0,2,4
```

14.1.8 Other useful array operations

```
1   # a is an array
2   a.clip(min=3, max=12) # clip elements
3   a.mean(); mean(a)     # mean value
4   a.var(); var(a)       # variance
5   a.std(); std(a)       # standard deviation
6   median(a)
7   cov(x,y)              # covariance
8   trapz(a)              # Trapezoidal integration
9   diff(a)               # finite differences (da/dx)
10
11  # more Matlab-like functions:
12  corrcoeff, cumprod, diag, eig, eye, fliplr, flipud, max, min, prod, ptp,
13  rot90, squeeze, sum, svd, tri, tril, triu
```

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)

# Calculate
tmp=numpy.array(list(map(lambda x: x.split(', '), lines)))
numb=numpy.array(tmp[:,1:], float).sum(axis=1)

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

```

```
f(5)
# 123
f(-1)
# -3
f = checkrange(f)
f(5)
# 123
f(-1)
# out of range
```

15.2.1 @decorator

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

g(2)
# 6
g(-2)
# out of range
```

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*

```
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*;

```
    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

lambda <args>: <body>
Example:

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

18 Map

map(function, iterable, ...)
Example:

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

19 List-comprehension

```
[f(x) for x in iterable] # returns a list
```

20 Dictionary-comprehension

```
{key: f(value) for x in dictionary} # returns a dictionary
```

21 Mixed language programming

- Suppose we have a C function:
  extern double hw1(double r1, double r2);
- We want to call this from Python as:

```
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):

```
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;

```
cdef int i, j, k
cdef float f, g[42], *h

```

- Function arguments and return types may be declared;

```
    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;

```
    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

```
  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

```
  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

```
<HTML><BODY BGCOLOR="white">
<FORM ACTION="hw1.py.cgi" METHOD="POST">
Hello, World! The sine of
<INPUT TYPE="text" NAME="r" SIZE="10" VALUE="1.2">
<INPUT TYPE="submit" VALUE="equals" NAME="equalsbutton">
</FORM></BODY></HTML>
```

CGI script

```
#!/store/bin/python
import cgi, math

# required opening of all CGI scripts with output:
print("Content-type: text/html\n")

# extract the value of the variable "r":
form = cgi.FieldStorage()
r=form.getvalue("r")

s=str(math.sin(float(r)))
# print answer (very primitive HTML code):
print("Hello, World! The sine of %s equals %s" % (r,s))
```

The complete improved CGI script

```
#!/store/bin/python
import cgi, math
print("Content-type: text/html\n") # std opening

# extract the value of the variable "r":
form = cgi.FieldStorage()
r=form.getvalue('r')
if r is not None:
  s=str(math.sin(float(r)))
else:
  s='';r=''

# print complete form with value:
print("""
<HTML><BODY BGCOLOR="white">
<FORM ACTION="hw2.py.cgi" METHOD="POST">
Hello, World! The sine of
<INPUT TYPE="text" NAME="r" SIZE="10" VALUE="%s">
<INPUT TYPE="submit" VALUE="equals" NAME="equalsbutton">
%s </FORM></BODY></HTML>\n""" % (r,s))
```

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
  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