Practical Python:

3: Intermediate to difficult exercises

Dafydd Gibbon

First South African Workshop in Digital Humanities
North-Western University, Potchefstroom, SA
2015-04-04 to 2015-04-05

Reminder: typical Python idioms - 1

- The **`if`** statement, which conditionally executes a block of code, along with **`else`** and **`elif`** (a contraction of **`else-if`**).

- The **`for`** statement, which iterates over an iterable object, capturing each element to a local variable for use by the attached block.

- The **`while`** statement, which executes a block of code as long as its condition is true.

- The **`try...except...`** statement, which allows exceptions raised in its attached code block to be caught and handled by except clauses; it also ensures that clean-up code in a finally block will always be run regardless of how the block exits.

Reminder: typical Python idioms - 2

- The **`class`** statement, which executes a block of code and attaches its local namespace to a class, for use in object-oriented programming.

- The **`def`** statement, which defines a function or method.

- The **`with`** statement (from Python 2.5), which encloses a code block within a context manager (for example, acquiring a lock before the block of code is run and releasing the lock afterwards, or opening a file and then closing it), allowing RAII-like behavior.

- The **`pass`** statement, which serves as a NOP. It is syntactically needed to create an empty code block.

Reminder: typical Python idioms - 3

- The **`assert`** statement, used during debugging to check for conditions that ought to apply.

- The **`yield`** statement, which returns a value from a generator function. From Python 2.5, yield is also an operator. This form is used to implement coroutines.

- The **`import`** statement, which is used to import modules whose functions or variables can be used in the current program.

- The **`print`** statement was changed to the **`print()`** function in Python 3.

Reminder: some Python objects

- Simple data types
  - characters
    - 1-char strings
  - numbers:
    - integers
    - floats

- Complex data types
  - iterable:
    - strings
    - lists (mutable)
      - recursive
  - non-iterable:
    - sets (mutable)
      - recursive
    - dictionaries (mutable)
      - recursive

- Functions and methods
  - built-in
    - pure
    - side-effects
  - home-made

- Local modules
  - components of applications

- Libraries
  - NumPy
  - MatplotLib
  - NLTK
  - and many others

Data types: strings

Official Python Tutorial:
https://docs.python.org/2/tutorial/

Extending strings:

```
textstring = '"Good morning!"'
textstring += ' said John.'
textstring = textstring + ' She said.'

storystring = textstring + """It was late in
the evening.
Next morning everything looked different.
The sky was blue. Birds were singing."""
```

Save to file:

```
open(filename,'w').write(storystring)
```

Exercise:

Write a routine for entering a story sentence by sentence, then saving it. Then read the sentence.

Data types: lists

CLI interaction: input

Official Python Tutorial:

https://docs.python.org/2/tutorial/

Basic interaction:

```
raw_input('Write something: ')
myinput = raw_input('Write something: ')

myinput = myinput.upper()
tokens = myinput.split(' ')

typeset = set(tokens)
typelist = list(typeset)
typelex = sorted(list(set(tokens)))

flist = [tokens.count(x) for x in typelex]
flist.reverse()
```

CLI interaction: input and output

Official Python Tutorial:

https://docs.python.org/2/tutorial/

Content:

```
textstring = 'Hello world!'
integernumber = 183
floatingpointnumber = 183.2345
floatingpointnumber = 1.0 * integernumber
floatingpointnumber = float(integernumber)
```

Filename input from command line as string:

```
filename = raw_input('Filename?')
```

Input from file as text string:

```
file_text = open(filename,'r').read()
```

Input from file as list of strings

```
file_linelist = open(filename,'r').readlines()
```

Output string to file:

```
open(filename,'w').write(textstring)
```

Control structures: conditions and Loops

Official Python Tutorial:

https://docs.python.org/2/tutorial/

Conditions:

```
if x == 1:
    return x
elif x == 2:
    return 0
else:
    return -1
```

Loops:

```
for item in list01:
    print item
    if item = 'rubbish':
        break
while item not in ['rubbish', 'nonsense']
    print item
```

Math and logic operators

Official Python Tutorial:

https://docs.python.org/2/tutorial/

- Arithmetic Operators
  + - * / % ** //
- Comparison (Relational) Operators
  == != <> < > <= >=
- Assignment Operators
  = += -= *= /= %= **= //=
- Logical Operators
  and or not
- Bitwise Operators
  & (AND) | (OR) ^ (XOR) ~ (bit flip) << l-shift >> r-shift
- Membership Operators
  in not in
- Identity Operators
  is is not

String operators

Official Python Tutorial:

https://docs.python.org/2/tutorial/

- String Operators
  + (concatenation) * (repetition) / %
- Slice, Range Slice
  [ ] [:]
- Assignment Operators
  = +=
- Membership Operators
  in not in
- String type Operators
  ' ' " " """ """ (multi-line)
  - r" " (raw, no escaping) u" " (unicode)

- String Formatting Operator
  %

String functions and methods

capitalize()
center(width, fillchar)
count(str, beg=0,end=len(string))
decode(encoding='UTF-8',errors='strict')
encode(encoding='UTF-8',errors='strict')
endswith(suffix, beg=0, end=len(string))
expandtabs(tabsize=8)
find(str, beg=0 end=len(string))
index(str, beg=0, end=len(string))
isalnum()
isalpha()
isdigit()
islower()
isnumeric()
isspace()
istitle()
isupper()
en(string)
ljust(width[, fillchar])

lower()
lstrip()
maketrans()
max(str)
min(str)
replace(old, new [, max])
rfind(str, beg=0,end=len(string))
rjust(width,[, fillchar])
rstrip()
split(str="", num=string.count(str))
splitlines(num=string.count('\n'))
startswith(str, beg=0,end=len(string))
strip()
swapcase()
title()
translate(table, deletechars="")
upper()
zfill (width)
isdecimal()

Official Python Tutorial:

https://docs.python.org/2/tutorial/

What does this do?

First analyse, then test, in the interactive environment:

```
a = 'a red one a blue one a yellow one and a banana'

b = a.strip().split()

sorted([(b.count(p),p) for p in set(b)], reverse=True)
```

What does this do?

First analyse, then test, in the interactive environment:

```
a = 'a red one a blue one a yellow one and a banana'

b = a.strip().split()

sorted([(b.count(p),p) for p in set(b)], reverse=True)

[(4, 'a'), (3, 'one'), (1, 'yellow'), (1, 'red'), (1,
'blue'), (1, 'bat'), (1, 'and')]
```

What does this do?

First analyse, then test, in the interactive environment:

```
a = 'a red one a blue one a yellow one and a banana'

b = a.strip().split()

sorted([(b.count(p),p) for p in set(b)], reverse=True)

[(4, 'a'), (3, 'one'), (1, 'yellow'), (1, 'red'), (1,
'blue'), (1, 'bat'), (1, 'and')]

b
```

What does this do?

First analyse, then test, in the interactive environment:

```
a = 'a red one a blue one a yellow one and a banana'

b = a.strip().split()

sorted([(b.count(p),p) for p in set(b)], reverse=True)

[(4, 'a'), (3, 'one'), (1, 'yellow'), (1, 'red'), (1,
'blue'), (1, 'bat'), (1, 'and')]

b

['a', 'red', 'one', 'a', 'blue', 'one', 'a', 'yellow',
'one', 'and', 'a', 'banana']
```

What does this do?

Copy the code into an editor in the right-hand window,
make sure you can print the result,
save as ‘banana.py’:

```
a = 'a red one a blue one a yellow one and a banana'

b = a.strip().split()

c = sorted([(b.count(p),p) for p in set(b)], reverse=True)

print c
```

Run the code:

```
python banana.py
```

What does this do?

Copy the code into an editor in the right-hand window,
make sure you can print the result,
save as ‘banana.py’:

```
a = 'a red one a blue one a yellow one and a banana'

b = a.strip().split()

c = sorted([(b.count(p),p) for p in set(b)], reverse=True)

print c
```

Run the code:

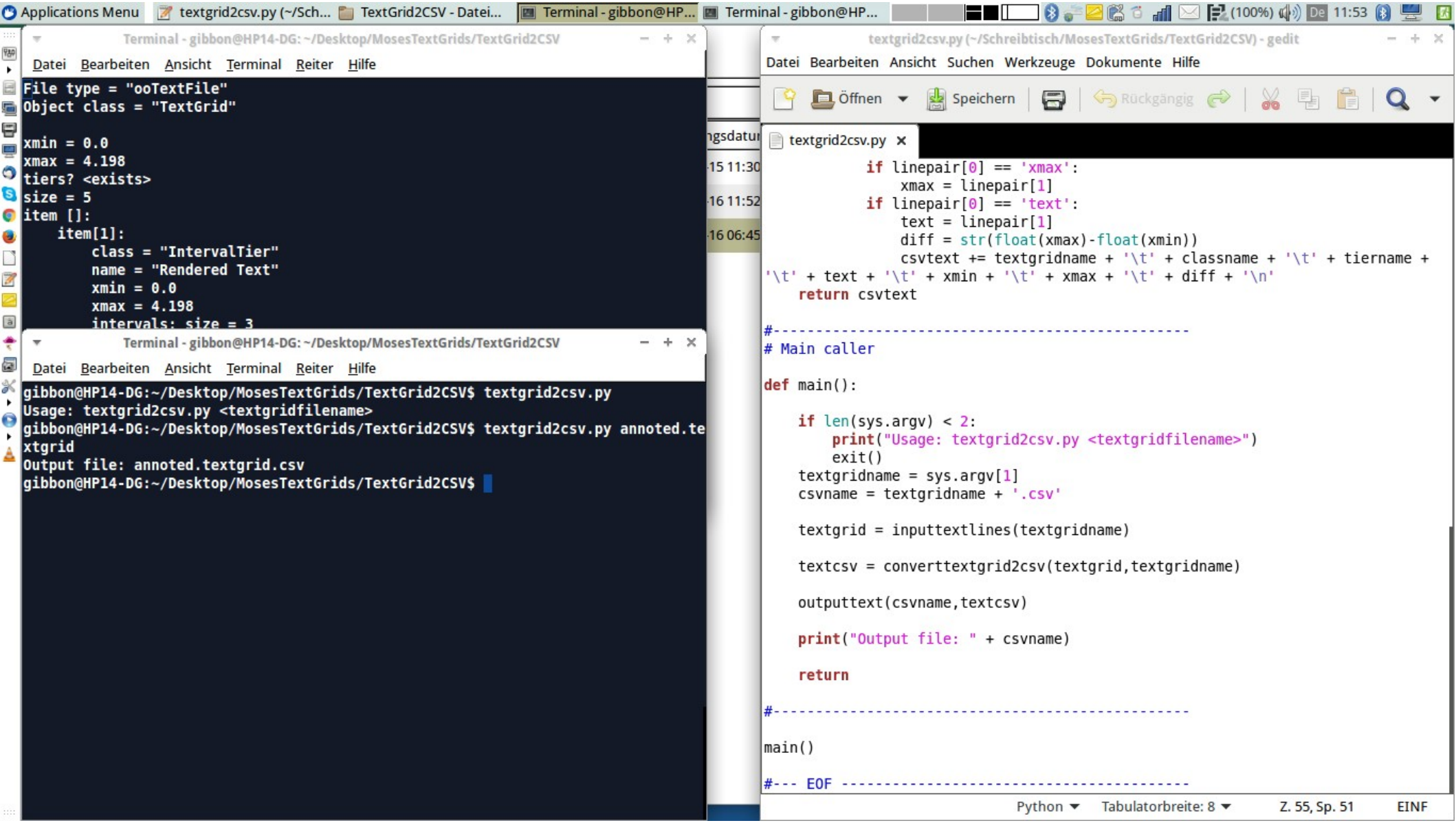
```
python banana.py

[(4, 'a'), (3, 'one'), (1, 'yellow'), (1, 'red'), (1,
'blue'), (1, 'banana'), (1, 'and')]
```

CLI workspace for slightly less retro work

Data window

Editor window



Runtime window

Now use an editor!

Add the ‘shebang line’ (a Unix / Linux feature) as first line, and (at least) minimal documentation:

```
#!/usr/bin/python
# banana.py
# D. Gibbon 2016.03.19
# Word frequency list demo

a = 'a red one a blue one a yellow one and a banana'

b = a.strip().split()

c = sorted([(b.count(p),p) for p in set(b)],reverse=True)

print c
```

Make the file executable, run the code directly, no python call:

```
chmod 755 banana.py
banana.py
[(4, 'a'), (3, 'one'), (1, 'yellow'), (1, 'red'), (1,
'blue'), (1, 'banana'), (1, 'and')]
```

Functions

Python functions

- Functions and arguments:

```
def foo(bar,baz):
    return bar + baz
```

- Error trapping as a test:

```
def  faa(bar,baz):
    try:
        return bar + ` ` + baz + `!`
    except:
        return bar + baz
```

- args vs. kwargs:

```
def fileext(string, extension='txt'):
    return string + '.' + extension
```

List comprehensions

- List comprehensions are iterative operators which map lists to lists

```
list1 = [1,2,-3,4,-5]
list2 = [x**2 for x in list1]
list3 = [0 if x < 0 else x**2 for x in list1[2:]]
```

- Embedded list comprehensions (beware)

```
a = [[1, 2], [3, 4]]
c = [x for b in a for x in b]      # [1, 2, 3, 4]
d = [x for x in b for b in a]      # [3, 3, 4, 4]
```

Note: The variable **b** retains its value outside the list comprehension.

List comprehensions

- Implement this formula $\frac{\sum_{i=1}^{n} d_i}{n}$
- with list variable `lst = [2,2,4,4,2]` as
  - list comprehension
  - for loop
  - while loop

List comprehensions

- Implement this formula $\frac{\sum_{i=1}^{n} d_i}{n}$
- with list variable `lst = [2,2,4,4,2]` as
  - list comprehension
  - for loop
  - while loop

You can do it like this, of course:

```
sum(lst)/float(len(lst))
```

But go ahead and use iterations...

List comprehensions

- Implement this formula $\frac{\sum_{i=1}^{n} d_i}{n}$
- with list variable `lst = [2,2,4,4,2]` as
  - list comprehension

```
float(sum([x for x in lst]))/len(lst)
```

  - for loop
  - while loop

List comprehensions

- Implement this formula $\frac{\sum_{i=1}^{n} d_i}{n}$
- with list variable `lst = [2,2,4,4,2]` as
  - list comprehension

```
float(sum([x for x in lst]))/len(lst)
```

  - for loop

```
s = 0.0
for i in lst:
    s += i
s/len(lst)
```

```
s = 0.0
for i in range(len(lst)):
    s += lst[i]
s/len(lst)
```

  - while loop

List comprehensions

- Implement this formula $\frac{\sum_{i=1}^{n} d_i}{n}$
- with list variable `lst = [2,2,4,4,2]`
  - list comprehension

```
float(sum([x for x in lst]))/len(lst)
```

  - for loop

```
s = 0.0
for i in lst:
    s += i
s/len(lst)
```

```
s = 0.0
for i in range(len(lst)):
    s += lst[i]
s/len(lst)
```

  - while loop

```
s = 0.0
ls = lst[:]
while ls:
    s += ls[0]
    ls = ls[1:]
s/len(lst)
```

Reverse engineering: loops, functions, list comprehensions

- Can you work out the formula behind this expresion?

```
lst = [2,4,2,4,2,4,2,4]

npvi = int(round(100*sum([0 if di - dj == 0 else
abs(di - dj)/((di + dj)/2.0) for di,dj in
zip(lst[:-1], lst[1:])]) / len(lst[:-1])))
```

Reverse engineering: loops, functions, list comprehensions

- Can you work out the formula behind this expresion?

```
lst = [2,4,2,4,2,4,2,4]

npvi = int(round(100*sum([0 if di - dj == 0 else
abs(di - dj)/((di + dj)/2.0) for di,dj in
zip(lst[:-1], lst[1:])]) / len(lst[:-1])))
```

Hints:

The value of **`npvi`** in this case is 67

Check the component function zip

Format the expression so that each bracket is embedded
(best using an editor rather than the Python interface)

And how to format the math formula in LibreOffice:

```
100*{sum_{i=1}^{n-1} {{d_i-d_{i_1}} over {(d_i+d_{i+1})/2} } over {n-1}}
```

Reverse engineering: loops, functions, list comprehensions

- Reformatted to reflect the structure of the formula:

```
int(round
      (
       100 * sum(
         [0
            if di - dj == 0
            else abs(di - dj) /((di + dj)/2.0)
              for di,dj in zip(lst[:-1], lst[1:])
          ]
         ) / len(lst[:-1])
        )
       )
```

Reverse engineering: loops, functions, list comprehensions

- One possible answer of several numerically equivalent answers:

$$100 * \frac{\sum_{i=1}^{n-1} \frac{\left| d_i - d_{i_1} \right|}{\left(d_i + d_{i+1} \right)/2}}{n-1}$$

The format as a LibreOffice formula object:

```
100*{sum_{i=1}^{n-1} {{d_i-d_{i_1}} over {(d_i+d_{i+1})/2}} over {n-1}}
```

More reverse engineering – what does this function do?

```
def mystery_function(a,b):

   c = {}
   n = len(a); m = len(b)

   for i in range(0,n+1):
      c[i,0] = i
   for j in range(0,m+1):
      c[0,j] = j

   for i in range(1,n+1):

      for j in range(1,m+1):
         x = c[i-1,j]+1
         y = c[i,j-1]+1
         if a[i-1] == b[j-1]:
            z = c[i-1,j-1]
         else:
            z = c[i-1,j-1]+1
         c[i,j] = min(x,y,z)

   return c[n,m]
```

More reverse engineering – what does this function do?

```
def mystery_function(a,b):

    c = {}
    n = len(a); m = len(b)

    for i in range(0,n+1):
        c[i,0] = i
    for j in range(0,m+1):
        c[0,j] = j

    for i in range(1,n+1):

        for j in range(1,m+1):
            x = c[i-1,j]+1
            y = c[i,j-1]+1
            if a[i-1] == b[j-1]:
                z = c[i-1,j-1]
            else:
                z = c[i-1,j-1]+1
            c[i,j] = min(x,y,z)

    return c[n,m]
```

Hint:

You will need this function later for some types of classification in Machine Learning.

Levenshtein Edit Distance Function (iterative style)

```
def levenshtein_iterative(a,b):

    c = {}
    n = len(a); m = len(b)

    for i in range(0,n+1):
        c[i,0] = i
    for j in range(0,m+1):
        c[0,j] = j

    for i in range(1,n+1):

        for j in range(1,m+1):
            x = c[i-1,j]+1
            y = c[i,j-1]+1
            if a[i-1] == b[j-1]:
                z = c[i-1,j-1]
            else:
                z = c[i-1,j-1]+1
            c[i,j] = min(x,y,z)

    return c[n,m]
```

Levenshtein edit distance, recursive style

```
Input: strings a and b

def levenshtein_fun(a, b):

    if not a: return len(b)
    if not b: return len(a)

    return min(int(a[0] != b[0]) +
        levenshtein_fun(a[1:], b[1:]), 1 +
        levenshtein_fun(a[1:], b), 1 +
        levenshtein_fun(a, b[1:]))
```

Some tricks with boolean and arithmetic values are used here. What are they?

We used to do this with LISP in the old days.

Levenshtein edit distance, recursive style

```
Input: strings a and b

def levenshtein_fun(a, b):

    if not a: return len(b)
    if not b: return len(a)

    return min(int(a[0] != b[0]) +
        levenshtein_fun(a[1:], b[1:]), 1 +
        levenshtein_fun(a[1:], b), 1 +
        levenshtein_fun(a, b[1:]))
```

if len(a) == 0: return len(b)
if len(b) == 0: return len(a)

if a[0]!=b[0]: return 1
else: return 0

substitution
deletion
insertion

Some tricks with boolean and arithmetic values are used here. What are they?

We used to do this with LISP in the old days.

Function application: `lambda`

- Lambda:
  - define an anonymous, non-persistent function:
    ```
    lambda x: x**0.5
    ```
    (from ‘lambda abstraction’ in formal logic)
  - apply to an argument:
    ```
    (lambda x: x**0.5)(9)
    ```
  - assign to a variable to yield a named, persistent function:
    ```
    sqrt = lambda x: x**0.5
    sqrt(9)
    ```
  - which is equivalent to a standard, stored, persistent function:
    ```
    def sqrt(x):
        return x**0.5
    sqrt(9)
    ```

- Try this out, and explain:
    ```
    for i in range(5):
        print i, (lambda x: x**i)(2.0)
    ```

Function application: `map`

- Map:
  - explicitly apply a named function to a sequence of its arguments (a tuple or a list):

```
list01 = [2, 3,  4, 5, 6]
map(float, list01)
```

  - apply an anonymous function to a sequence of its arguments:

```
map((lambda x: x * 1.0), list01)
```

  - for more than one argument, to sequences for each argument:

```
list02 = [3, 4, 5, 6, 7]
map((lambda x, y: x + y), list01, list02)
```

Function application: `filter`, `reduce`

Filter:

- select specified items from list:

```
list03 = [2, 3, 4, 'five', 'six']
filter((lambda x: isinstance(x,str)), list03)
filter((lambda x: type(x) == str), list03)
filter((lambda x: type(x) != str), list03)
```

Reduction:

- apply a function to all members of the list in turn, applying each value to the next item:

```
def add(i,j):
    return i + j

reduce(add,[0,1,2,3,4])
reduce((lambda x, y: x + y), range(5))
```

Function application vs. list comprehension

```
lst = [1,2,3,4,5]

a = [float(x) for x in lst]

b = map(float,lst)

c = map(lambda x: float(x),  lst)
```

Contrast:

```
d = filter(float,lst)
e = filter(lambda x: float(x), lst)
```

Importing libraries

The main libraries which I use

NumPy: numerical calculation

```
import numpy as np
```

SciPy: scientific calculation

```
from scipy import ...
```

SciKit-Learn: machine learning

```
from sklearn import ...
```

MatPlotLib: many kinds of line diagram graphics

```
import matplotlib as mpl
import matplotlib.pyplot as plt
```

NetworX: calculation with networks (with GraphViz)

```
import networkx as nx
```

Other libraries which I use frequently

Regular expression handling

```
import re
```

System calls

```
import sys, os, traceback, inspect
import time
```

Networking

```
import socket
```

Server-side Web apps

```
import cgitb; cgitb.enable()
```

Modules – your own libraries

- Code for any frequently used functionality can be re-used just like any other library, as a module.

- For example:

```
def errormessage(string):
    print 'Error message:', string
    return
```

- Save as **`errhandler.py`**

- Then

```
import errhandler.py as eh
  – eh.errmessage('Sorry, that did not work.')
```

Importing libraries

NumPy: mathematical library

- arrays with matrix operations, efficien implementations

Official Python Tutorial:

https://docs.python.org/2/tutorial/

```
import numpy
x = numpy.sqrt(25)

import numpy as np
x = np.sqrt(25)

from numpy import sqrt
x = sqrt(25)

a = [[1,2,3,4],[3,4,5,6]]
b = np.array([[1,2,3,4],[3,4,5,6]])
```

- `a * a` (throws an error)
- `b * b (yields the product of corresponding cells)`

Assignment:

Find and plot time series data on the web (e.g. some value per day, of month, year etc.).

Assignment: speech annotation to CSV task (not too easy)

```
File type = "ooTextFile"
Object class = "TextGrid"

xmin = 0.0
xmax = 4.198
tiers? <exists>
size = 5
item []:
    item[1]:
        class = "IntervalTier"
        name = "Rendered Text"
        xmin = 0.0
        xmax = 4.198
        intervals: size = 3
        intervals [1]
            xmin = 0.0
            xmax = 0.4
            text = ""
        intervals [2]
            xmin = 0.4
            xmax = 3.28
            text = "Àbàsì ámáà síák úsʌ́ñ úbò̩kkó ánọ̀ ǹdìtọ̀ Ísrěd"
        intervals [3]
            xmin = 3.28
            xmax = 4.198
            text = ""
    item[2]:
        class = "IntervalTier"
        name = "syllable"
        xmin = 0.0
        xmax = 4.198
        intervals: size = 34
        intervals [1]
            xmin = 0.0
            xmax = 0.36
            text = ""
```

The format is Praat TextGrid (check on the internet).

Required output fields in the CSV file:

- filename
- tiername
- label
- start
- end
- duration

Now calculate average duration and standard deviation of duration.

End of Unit 3