Week 4

Sun Jun

with slides from Hans Petter Langtangen

Cohort 1: Input Data

Getting input from questions and anwsers

- Sample program:

```
C = 21
F = (9.0/5)*C + 32
print F
```

- Idea: let the program ask the user a question "C=?", read the user's answer, assign that answer to the variable `C`
- This is easy:

```
C = raw_input('C=? ')    # C becomes a string
C = float(C)
F = (9./5)*C + 32
print F
```

- Testing:

```
Unix/DOS> python c2f_qa.py
C=? 21
69.8
```

Print the n first even numbers

- Read `n` from the keyboard:

```
n = int(raw_input('n=? '))

for i in range(2, 2*n+1, 2):
    print i

# or:
print range(2, 2*n+1, 2)

# or:
for i in range(1, n+1):
    print 2*i
```

The magic eval function

- `eval(s)` evaluates a string object `s` as if the string had been written directly into the program
- Example `r = eval('1+1')` is the same as

```
r = 1+1
```

- Some other examples:

```
>>> r = eval('1+2')
>>> r
3
>>> type(r)
<type 'int'>

>>> r = eval('[1, 6, 7.5]')
>>> r
[1, 6, 7.5]
>>> type(r)
<type 'list'>
```

Be careful with eval and string values

- Task: write `r = eval(...)` that is equivalent to

```
r = 'math programming'
```

- Must use quotes to indicate that `'math programming'` is string, plus extra quotes:

```
r = eval("'math programming'")
# or
r = eval('"math programming"')
```

- What if we forget the extra quotes?

```
r = eval('math programming')
```

  is the same as

```
r = math programming
```

  but then Python thinks `math` and `programming` are two variables...combined with wrong syntax

A little program can do much

- This program adds two input variables:

```
i1 = eval(raw_input('operand 1: '))
i2 = eval(raw_input('operand 2: '))
r = i1 + i2
print '%s + %s becomes %s
with value %s' %       (type(i1), type(i2), type(r), r)
```

- We can add integer and float:

```
Unix/DOS> python add_input.py
operand 1: 1
operand 2: 3.0
<type 'int'> + <type 'float'> becomes <type 'float'>
with value 4
```

- We can add two lists:

```
Unix/DOS> python add_input.py
operand 1: [1,2]
operand 2: [-1,0,1]
<type 'list'> + <type 'list'> becomes <type 'list'>
with value [1, 2, -1, 0, 1]
```

- Note: `r = i1 + i2` becomes the same as

```
r = [1,2] + [-1,0,1]
```

A similar magic function: exec

- `eval(s)` evaluates an *expression* `s`
- `eval('r = 1+1')` is illegal because this is a statement, not only an expression (assignment statement: variable = expression)
- ...but we can use `exec` for complete statements:

```
statement = 'r = 1+1'    # store statement in a string
exec(statement)
print r
```

  will print 2
- For longer code we can use multi-line strings:

```
somecode = '''
def f(t):
    term1 = exp(-a*t)*sin(w1*x)
    term2 = 2*sin(w2*x)
    return term1 + term2
'''
exec(somecode)  # execute the string as Python code
```

What can exec be used for?

- Build code at run-time, e.g., a function:

```
formula = raw_input('Write a formula involving x: ')
code = """
def f(x):
    return %s
""" % formula
exec(code)

x = 0
while x is not None:
    x = eval(raw_input('Give x (None to quit): '))
    if x is not None:
        y = f(x)
        print 'f(%g)=%g' % (x, y)
```

- While the program is running, the user types a formula, which becomes a function, the user gives `x` values until the answer is `None`, and the program evaluates the function `f(x)`
- Note: the programmer knows nothing about `f(x)`!

Exercise

Write a function named inputVal(s), that takes an argument s (string type), applies eval to this input, and prints out the type of the resulting object and its value. Test your function by using five types of input: an integer, a real number, a complex number, a list and a tuple.

Reading from the command line

- Consider again our Celsius-Fahrenheit program:

```
C = 21; F = (9.0/5)*C + 32; print F
```

- Now we want to provide `C` as a command-line argument after the name of the program when we run the program:

```
Unix/DOS> python c2f_cml_v1.py 21
69.8
```

- Command-line arguments = "words" after the program name
- The list `sys.argv` holds the command-line arguments:

```
import sys
print 'program name:', sys.argv[0]
print '1st command-line argument:', sys.argv[1] # string
print '2nd command-line argument:', sys.argv[2] # string
print '3rd command-line argument:', sys.argv[3] # string
etc.
```

- The Celsius-Fahrenheit conversion program:

```
import sys
C = float(sys.argv[1])
F = 9.0*C/5 + 32
print F
```

How are command-line arguments separated?

- Command-line arguments are separated by blanks – use quotes to override this rule!
- Let us make a program for printing the command-line args.:

```
import sys; print sys.argv[1:]
```

- Demonstrations:

```
Unix/DOS> python print_cml.py 21 string with blanks 1.3
['21', 'string', 'with', 'blanks', '1.3']

Unix/DOS> python print_cml.py 21 "string with blanks" 1.3
['21', 'string with blanks', '1.3']
```

- Note that all list elements are surrounded by quotes, showing that command-line arguments are strings

Example on reading 3 parameters from the command line

- Compute the current location of an object,

$$s(t) = s_0 + v_0 t + \frac{1}{2} a t^2$$

  when s_0 (initial location), v_0 (initial velocity), a (constant acceleration) and t (time) are given on the command line

- How far away is the object at $t = 3$ s, if it started at $s_0 = 1$ m at $t = 0$ with a velocity $v_0 = 1$ m/s and has undergone a constant acceleration of 0.5 m/s^2?

```
Unix/DOS> python location_cml.py 1 1 0.5 3
6.25
```

- Program:

```
import sys
s0 = float(sys.argv[1])
v0 = float(sys.argv[2])
a  = float(sys.argv[3])
t  = float(sys.argv[4])
s  = s0 + v0*t + 0.5*a*t*t
print s
```

Command-line arguments with options

- Many programs, especially on Unix systems, take a set of command-line arguments of the form `--option value`

```
Unix/DOS> python location.py --v0 1 --t 3 --s0 1 --a 0.5
```

- Provide sensible default values
- Type just `--a 4 --t 8` if only the default values of `a` and `t` need to be changed
- More user-friendly than requiring a complete sequence of command-line arguments (like positional arguments vs keyword arguments)

Command-line arguments with options

- Can use the module getopt to help reading the data:

```
s0 = 0; v0 = 0; a = t = 1   # default values
import getopt, sys
options, args = getopt.getopt(sys.argv[1:], '',
                ['t=', 's0=', 'v0=', 'a='])

# options is a list of 2-tuples (option,value) of the
# option-value pairs given on the command line, e.g.,
# [('--v0', 1.5), ('--t', 0.1), ('--a', 3)]

for option, value in options:
    if option == '--t':
        t = float(value)
    elif option == '--a':
        a = float(value)
    elif option == '--v0':
        v0 = float(value)
    elif option == '--s0':
        s0 = float(value)
```

Multiple versions of command-line args (long and short)

- We can allow both long and shorter options, e.g. `--t` and `--time`, and `--a` and `--acceleration`

```
options, args = getopt.getopt(sys.argv[1:], '',
  ['v0=', 'initial_velocity=', 't=', 'time=',
   's0=', 'initial_velocity=', 'a=', 'acceleration='])

for option, value in options:
    if option in ('--t', '--time'):
        t = float(value)
    elif option in ('--a', '--acceleration'):
        a = float(value)
    elif option in ('--v0', '--initial_velocity'):
        v0 = float(value)
    elif option in ('--s0', '--initial_position'):
        s0 = float(value)
```

Summary of `--option value` pairs

- Advantage of `--option value` pairs:
  - can give options and values in arbitrary sequence
  - can skip option if default value is ok
- Command-line arguments that we read as `sys.argv[1]`, `sys.argv[2]`, etc. are like positional arguments to functions: the right sequence of data is essential!
- `--option value` pairs are like keyword arguments – the sequence is arbitrary and all options have a default value

Exercise

Consider the following program segment

```
v0 =3;g =9.81; t =0.6
y=v0*t -0.5* g*t**2
print y
```

Modify this program segment so that a function ballQa(t,v0) takes argument t and v0 from the command line and returns the value of y.

Reading data from a file

- A file is a sequence of characters (text)
- We can read text in the file into strings in a program
- This is a common way for a program to get input data
- Basic recipe:

```
infile = open('myfile.dat', 'r')
# read next line:
line = infile.readline()

# read lines one by one:
for line in infile:
    <process line>

# load all lines into a list of strings (lines):
lines = infile.readlines()
for line in lines:
    <process line>
```

Example: reading a file with numbers (part 1)

- The file `data1.txt` has a column of numbers:

```
21.8
18.1
19
23
26
17.8
```

- Goal: compute the average value of the numbers:

```
infile = open('data1.txt', 'r')
lines = infile.readlines()
infile.close()
mean = 0
for number in lines:
    mean = mean + number
mean = mean/len(lines)
```

- Running the program gives an error message:

```
TypeError: unsupported operand type(s) for +:
           'int' and 'str'
```

- Problem: `number` is a string!

Example: reading a file with numbers (part 2)

- We must convert strings to numbers before computing:

```
infile = open('data1.txt', 'r')
lines = infile.readlines()
infile.close()
mean = 0
for line in lines:
    number = float(line)
    mean = mean + number
mean = mean/len(lines)
print mean
```

- A quicker and shorter variant:

```
infile = open('data1.txt', 'r')
numbers = [float(line) for line in infile.readlines()]
infile.close()
mean = sum(numbers)/len(numbers)
print mean
```

While loop over lines in a file

Especially older Python programs employ this technique:

```
infile = open('data1.txt', 'r')
mean = 0
n = 0
while True:                    # loop "forever"
    line = infile.readline()
    if not line:               # line='' at end of file
        break                  # jump out of loop
    mean += float(line)
    n += 1
infile.close()
mean = mean/float(n)
print mean
```

Experiment with reading techniques

```
>>> infile = open('data1.txt', 'r')
>>> fstr = infile.read()        # read file into a string
>>> fstr
'21.8\n18.1\n19\n23\n26\n17.8\n'
>>> line = infile.readline()    # read after end of file...
>>> line
''
>>> bool(line)                  # test if line:
False                           # empty object is False
>>> infile.close(); infile = open('data1.txt', 'r')
>>> lines = infile.readlines()
>>> lines
['21.8\n', '18.1\n', '19\n', '23\n', '26\n', '17.8\n']
>>> infile.close(); infile = open('data1.txt', 'r')
>>> for line in infile: print line,
...
21.8
18.1
19
23
26
17.8
```

Reading a mixture of text and numbers (part 1)

- The file `rainfall.dat` looks like this:

```
Average rainfall (in mm) in Rome: 1188 months between 1782 and 1970
Jan  81.2
Feb  63.2
Mar  70.3
Apr  55.7
May  53.0
...
```

- Goal: read the numbers and compute the mean
- Technique: for each line, split the line into words, convert the 2nd word to a number and add to sum

```
for line in infile:
    words = line.split()     # list of words on the line
    number = float(words[1])
```

- Note `line.split()`: very useful for grabbing individual words on a line, can split wrt any string, e.g., `line.split(';')`, `line.split(':')`

Reading a mixture of text and numbers (part 2)

The complete program:

```
def extract_data(filename):
    infile = open(filename, 'r')
    infile.readline() # skip the first line
    numbers = []
    for line in infile:
        words = line.split()
        number = float(words[1])
        numbers.append(number)
    infile.close()
    return numbers

values = extract_data('rainfall.dat')
from scitools.std import plot
month_indices = range(1, 13)
plot(month_indices, values[:-1], 'o2')
```

What is a file?

- A file is a sequence of characters
- For simple text files, each character is one byte (=8 bits, a bit is 0 or 1), which gives $2^8 = 256$ different characters
- (Text files in, e.g., Chinese and Japanese need several bytes for each character)
- Save the text "ABCD" to file in Emacs and OpenOffice/Word and examine the file
- In Emacs, the file size is 4 bytes

Exercise

A file named xy.dat contains two columns of numbers, corresponding to the x and the y coordinates on a curve. The start of the file looks as follows.

```
-1.0000 -0.0000
-0.9933 -0.0087
-0.9867 -0.0179
-0.9800 -0.0274
-0.9733 -0.0374
```

Write a function named read2columns(f) with argument f of file object (e.g., f=open('xy.data','r')). The function should read the first column from the file into a list x and the second column into a list y. The function returns the maximum and minimum y coordinates.

File writing

- File writing is simple: collect the text you want to write in one or more strings and do, for each string, a

```
outfile.write(string)
```

- `outfile.write` does not add a newline, like `print`, so you may have to do that explicitly:

```
outfile.write(string + '\n')
```

- That's it! Compose the strings and write!

Example: writing a nested list (table) to file (part 1)

- Given a table like

```
data = \
[[ 0.75,        0.29619813, -0.29619813, -0.75      ],
 [ 0.29619813,  0.11697778, -0.11697778, -0.29619813],
 [-0.29619813, -0.11697778,  0.11697778,  0.29619813],
 [-0.75,       -0.29619813,  0.29619813,  0.75      ]]
```

- Write this nested list to a file

Example: writing a nested list to file (part 2)

```
outfile = open('tmp_table.dat', 'w')
for row in data:
    for column in row:
        outfile.write('%14.8f' % column)
    outfile.write('
')   # ensure linebreak
outfile.close()
```

Summary of file reading and writing

- Reading a file:

```
infile = open(filename, 'r')
for line in infile:
    # process line

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

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

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

infile.close()
```

- Writing a file:

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

Cohort 2: Exceptions and Strings

This great flexibility also quickly breaks programs...

```
Unix/DOS> python add_input.py
operand 1: (1,2)
operand 2: [3,4]
Traceback (most recent call last):
  File "add_input.py", line 3, in <module>
    r = i1 + i2
TypeError: can only concatenate tuple (not "list") to tuple

Unix/DOS> python add_input.py
operand 1: one
Traceback (most recent call last):
  File "add_input.py", line 1, in <module>
    i1 = eval(raw_input('operand 1: '))
  File "<string>", line 1, in <module>
NameError: name 'one' is not defined

Unix/DOS> python add_input.py
operand 1: 4
operand 2: 'Hello, World!'
Traceback (most recent call last):
  File "add_input.py", line 3, in <module>
    r = i1 + i2
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Handling errors in input

- A user can easily use our program in a wrong way, e.g.,

```
Unix/DOS> python c2f_cml_v1.py
Traceback (most recent call last):
  File "c2f_cml_v1.py", line 2, in ?
    C = float(sys.argv[1])
IndexError: list index out of range
```

  (the user forgot to provide a command-line argument...)

- How can *we* take control, explain what was wrong with the input, and stop the program without strange Python error messages?

```
if len(sys.argv) < 2:
    print 'You failed to provide a command-line arg.!'
    sys.exit(1)  # abort
F = 9.0*C/5 + 32
print '%gC is %.1fF' % (C, F)
```

- Execution:

```
Unix/DOS> python c2f_cml_v2.py
You failed to provide a command-line arg.!
```

Exceptions instead of if tests

- Rather than test "if something is wrong, recover from error, else do what we indended to do", it is common in Python (and many other languages) to *try* to do what we indend to, and if it fails, we recover from the error
- This principle makes use of a `try-except` block:

```
try:
    <statements we indend to do>
except:
    <statements for handling errors>
```

- If something goes wrong in the `try` block, Python raises an *exception* and the execution jumps immediately to the `except` block
- Let's see it in an example!

Celsius-Fahrenheit conversion with try-except

- Try to read `C` from the command-line, if it fails, tell the user and abort execution:

```
import sys
try:
    C = float(sys.argv[1])
except:
    print 'You failed to provide a command-line arg.!'
    sys.exit(1)  # abort
F = 9.0*C/5 + 32
print '%gC is %.1fF' % (C, F)
```

- Execution:

```
Unix/DOS> python c2f_cml_v3.py
You failed to provide a command-line arg.!

Unix/DOS> python c2f_cml_v4.py 21C
You failed to provide a command-line arg.!
```

Testing for a specific exception

- In

  ```
  try:
      <statements>
  except:
      <statements>
  ```

  we jump to the `except` block for *any* exception raised when executing the `try` block

- It is good programming style to test for specific exceptions:

  ```
  try:
      C = float(sys.argv[1])
  except IndexError:
      ...
  ```

- If we have an index out of bounds in `sys.argv`, an `IndexError` exception is raised, and we jump to the `except` block

- If any other exception arises, Python aborts the execution:

  ```
  Unix/DOS>> python c2f_cml_tmp.py 21C
  Traceback (most recent call last):
    File "tmp.py", line 3, in <module>
      C = float(sys.argv[1])
  ValueError: invalid literal for float(): 21C
  ```

Branching into different except blocks

- We can test for different exceptions:

```
import sys
try:
    C = float(sys.argv[1])
except IndexError:
    print 'No command-line argument for C!'
    sys.exit(1)  # abort execution
except ValueError:
    print 'Celsius degrees must be a pure number, '        'r
    sys.exit(1)

F = 9.0*C/5 + 32
print '%gC is %.1fF' % (C, F)
```

- Execution:

```
Unix/DOS> python c2f_cml_v3.py
No command-line argument for C!

Unix/DOS> python c2f_cml_v3.py 21C
Celsius degrees must be a pure number, not "21C"
```

The programmer can raise exceptions (part 1)

- Instead of just letting Python raise exceptions, we can raise our own and tailor the message to the problem at hand
- We provide two examples on this:
  - catching an exception, but raising a new one with an improved (tailored) error message
  - raising an exception because of wrong input data
- Example:

```
def read_C():
    try:
        C = float(sys.argv[1])
    except IndexError:
        raise IndexError        ('Celsius degrees must be supplied
    except ValueError:
        raise ValueError        ('Celsius degrees must be a pure nu
    # C is read correctly as a number, but can have wrong value:
    if C < -273.15:
        raise ValueError('C=%g is a non-physical value!' % C)
    return C
```

The programmer can raise exceptions (part 2)

- Calling the function in the main program:

```
try:
    C = read_C()
except (IndexError, ValueError), e:
    # print exception message and stop the program
    print e
    sys.exit(1)
```

- Examples on running the program:

```
Unix/DOS> c2f_cml.py
Celsius degrees must be supplied on the command line

Unix/DOS> c2f_cml.py 21C
Celsius degrees must be a pure number, not "21C"

Unix/DOS> c2f_cml.py -500
C=-500 is a non-physical value!

Unix/DOS> c2f_cml.py 21
21C is 69.8F
```

Exercise

Week 4, Cohort Session Problems, Question 4

String manipulation

- Text in Python is represented as strings
- Programming with strings is therefore the key to interpret text in files and construct new text
- First we show some common string operations and then we apply them to real examples
- Our sample string used for illustration is

```
>>> s = 'Berlin: 18.4 C at 4 pm'
```

- Strings behave much like lists/tuples - they are a sequence of characters:

```
>>> s[0]
'B'
>>> s[1]
'e'
```

Extracting substrings

- Substrings are just as slices of lists and arrays:

```
>>> s
'Berlin: 18.4 C at 4 pm'
>>> s[8:]     # from index 8 to the end of the string
'18.4 C at 4 pm'
>>> s[8:12]   # index 8, 9, 10 and 11 (not 12!)
'18.4'
>>> s[8:-1]
'18.4 C at 4 p'
>>> s[8:-8]
'18.4 C'
```

- Find start of substring:

```
>>> s.find('Berlin')  # where does 'Berlin' start?
0                     # at index 0
>>> s.find('pm')
20
>>> s.find('Oslo')    # not found
-1
```

Checking if a substring is contained in a string

```
>>> 'Berlin' in s:
True
>>> 'Oslo' in s:
False

>>> if 'C' in s:
...     print 'C found'
... else:
...     print 'no C'
...
C found
```

Substituting a substring by another string

- `s.replace(s1, s2)`: replace `s1` by `s2`

```
>>> s.replace(' ', '__')
'Berlin:__18.4__C__at__4__pm'
>>> s.replace('Berlin', 'Bonn')
'Bonn: 18.4 C at 4 pm'
```

- Example: replacing the text before the first colon by `'Bonn'`

```
\>>> s
'Berlin: 18.4 C at 4 pm'
>>> s.replace(s[:s.find(':')], 'Bonn')
'Bonn: 18.4 C at 4 pm'
```

- 1) `s.find(':')` returns 6, 2) `s[:6]` is `'Berlin'`, 3) this is replaced by `'Bonn'`

Splitting a string into a list of substrings

- Split a string into a list of substrings where the seperator is `sep`: `s.split(sep)`
- No separator implies split wrt whitespace

```
>>> s
'Berlin: 18.4 C at 4 pm'
>>> s.split(':')
['Berlin', ' 18.4 C at 4 pm']
>>> s.split()
['Berlin:', '18.4', 'C', 'at', '4', 'pm']
```

- Try to understand this one:

```
>>> s.split(':')[1].split()[0]
'18.4'
>>> deg = float(_)  # convert last result to float
>>> deg
18.4
```

Splitting a string into lines

- Very often, a string contains lots of text and we want to split the text into separate lines
- Lines may be separated by different control characters on different platforms. On Unix/Linux/Mac, backslash n is used:

```
>>> t = '1st line\n2nd line\n3rd line'
>>> print t
1st line
2nd line
3rd line
>>> t.split('\n')
['1st line', '2nd line', '3rd line']
>>> t.splitlines()  # cross platform - better!
['1st line', '2nd line', '3rd line']
```

Strings are constant (immutable) objects

- You cannot change a string in-place (as you can with lists and arrays) - all changes of a strings results in a new string

```
>>> s[18] = 5
...
TypeError: 'str' object does not support item assignment

>>> # build a new string by adding pieces of s:
>>> s[:18] + '5' + s[19:]
'Berlin: 18.4 C at 5 pm'
```

Stripping off leading/trailing whitespace

```
>>> s = '   text with leading/trailing space   \n'
>>> s.strip()
'text with leading/trailing space'
>>> s.lstrip()   # left strip
'text with leading/trailing space   \n'
>>> s.rstrip()   # right strip
'   text with leading/trailing space'
```

Some convenient string functions

```
>>> '214'.isdigit()
True
>>> '  214 '.isdigit()
False
>>> '2.14'.isdigit()
False

>>> s.lower()
'berlin: 18.4 c at 4 pm'
>>> s.upper()
'BERLIN: 18.4 C AT 4 PM'

>>> s.startswith('Berlin')
True
>>> s.endswith('am')
False

>>> '    '.isspace()   # blanks
True
>>> '  \n'.isspace()   # newline
True
>>> '  \t '.isspace()  # TAB
True
>>> ''.isspace()       # empty string
False
```

Joining a list of substrings to a new string

- We can put strings together with a delimiter in between:

```
>>> strings = ['Newton', 'Secant', 'Bisection']
>>> ', '.join(strings)
'Newton, Secant, Bisection'
```

- These are inverse operations:

```
t = delimiter.join(stringlist)
stringlist = t.split(delimiter)
```

- Split off the first two words on a line:

```
>>> line = 'This is a line of words separated by space'
>>> words = line.split()
>>> line2 = ' '.join(words[2:])
>>> line2
'a line of words separated by space'
```

Exercise

- Dene a function reverse(s) that computes the reversal of a string. For example,

    reverse("I am testing")

should return the string

    “gnitset ma I”

Example: read pairs of numbers (x,y) from a file

- Sample file:

```
(1.3,0)     (-1,2)     (3,-1.5)
(0,1)       (1,0)      (1,1)
(0,-0.01)   (10.5,-1)  (2.5,-2.5)
```

- Method: read line by line, for each line: split line into words, for each word: split off the parethesis and the split the rest wrt comma into two numbers

The code for reading pairs

```
lines = open('read_pairs.dat', 'r').readlines()

pairs = []   # list of (n1, n2) pairs of numbers
for line in lines:
    words = line.split()
    for word in words:
        word = word[1:-1]  # strip off parenthesis
        n1, n2 = word.split(',')
        n1 = float(n1);  n2 = float(n2)
        pair = (n1, n2)
        pairs.append(pair)  # add 2-tuple to last row
```

Output of a pretty print of the pairs list

```
[(1.3, 0.0),
 (-1.0, 2.0),
 (3.0, -1.5),
 (0.0, 1.0),
 (1.0, 0.0),
 (1.0, 1.0),
 (0.0, -0.01),
 (10.5, -1.0),
 (2.5, -2.5)]
```

Alternative solution: Python syntax in file format

- What if we write, in the file, the pairs (x,y) with comma in between the pairs?
- Adding a leading and trailing square bracket gives a Python syntax for a list of tuples (!)
- `eval` on that list could reproduce the list...
- The file format:

```
(1.3, 0),    (-1, 2),    (3, -1.5)
...
```

- We want to add a comma at the end of every line and square brackets around the whole file text, and then do an eval:

```
list = eval("[(1.3,0),    (-1, 2),    (3, -1.5),
...
]")
```

The code for reading pairs with eval

```
infile = open('read_pairs_wcomma.dat', 'r')
listtext = '['
for line in infile:
    # add line, without newline (line[:-1]),
    # with a trailing comma:
    listtext += line[:-1] + ', '
infile.close()
listtext = listtext + ']'
pairs = eval(listtext)
```

Exercise

Write a function named getBaseCounts( ) that takes a DNA string as input. The input string consists of letters A, C, T, and G. The function prints the number of A, C, T and G in the string. For instance, Given

“AAGCTAAGCCTGA”

It prints: A*5, C*3, G*3, T*2

Cohort 3: Dictionary

Dictionaries

- Lists and arrays are fine for collecting a bunch of objects in a single object
- List and arrays use an integer index, starting at 0, for reaching the elements
- For many applications the integer index is "unnatural" - a general text (or integer not restricted to start at 0) will ease programming
- Dictionaries meet this need
- Dictionary = list with text (or any constant object) as index
- Other languages use names like hash, HashMap and associative array for what is known as dictionary in Python

Example on a dictionary

- Suppose we need to store the temperatures in Oslo, London and Paris
- List solution:

```
temps = [13, 15.4, 17.5]
# temps[0]: Oslo
# temps[1]: London
# temps[2]: Paris
```

- We need to remember the mapping between the index and the city name – with a dictionary we can index the list with the city name directly (e.g., `temps["Oslo"]`):

```
temps = {'Oslo': 13, 'London': 15.4, 'Paris': 17.5}
# or
temps = dict(Oslo=13, London=15.4, Paris=17.5)
# application:
print 'The temperature in London is', temps['London']
```

Dictionary operations (part 1)

- Add a new element to a dict (dict = dictionary):

```
>>> temps['Madrid'] = 26.0
>>> print temps
{'Oslo': 13, 'London': 15.4, 'Paris': 17.5,
 'Madrid': 26.0}
```

- Loop (iterate) over a dict:

```
>>> for city in temps:
...     print 'The temperature in %s is %g' % \
...           (city, temps[city])
...
The temperature in Paris is 17.5
The temperature in Oslo is 13
The temperature in London is 15.4
The temperature in Madrid is 26
```

- The index in a dictionary is called *key*
  (a dictionary holds key–value pairs)

```
for key in dictionary:
    value = dictionary[key]
    print value
```

Dictionary operations (part 2)

- Does the dict have a particular key?

```
>>> if 'Berlin' in temps:
...     print 'Berlin:', temps['Berlin']
... else:
...     print 'No temperature data for Berlin'
...
No temperature data for Berlin
>>> 'Oslo' in temps     # standard boolean expression
True
```

- The keys and values can be reached as lists:

```
>>> temps.keys()
['Paris', 'Oslo', 'London', 'Madrid']
>>> temps.values()
[17.5, 13, 15.4, 26.0]
```

- Note: the sequence of keys is arbitrary! Never rely on it – if you need a specific order of the keys, use a sort:

```
for key in sorted(temps):
    value = temps[key]
    print value
```

Dictionary operations (part 3)

- More operations:

```
>>> del temps['Oslo']    # remove Oslo key w/value
>>> temps
{'Paris': 17.5, 'London': 15.4, 'Madrid': 26.0}
>>> len(temps)           # no of key-value pairs in dict.
3
```

- Two variables can refer to the same dictionary:

```
>>> t1 = temps
>>> t1['Stockholm'] = 10.0     # change t1
>>> temps                      # temps is also changed
{'Stockholm': 10.0, 'Paris': 17.5, 'London': 15.4,
  'Madrid': 26.0}
>>> t2 = temps.copy()          # take a copy
>>> t2['Paris'] = 16
>>> t1['Paris']
17.5
```

Examples: polynomials represented by dictionaries

- The polynomial

$$p(x) = -1 + x^2 + 3x^7$$

  can be represented by a dict with power as key and coefficient as value:

```
p = {0: -1, 2: 1, 7: 3}
```

- Evaluate polynomials represented as dictionaries: $\sum_{i \in I} c_i x^i$

```
def poly1(data, x):
    sum = 0.0
    for power in data:
        sum += data[power]*x**power
    return sum
```

- Shorter:

```
def poly1(data, x):
    return sum([data[p]*x**p for p in data])
```

Lists as dictionaries

- A list can also represent a polynomial
- The list index must correspond to the power
- $-1 + x^2 + 3x^7$ becomes

```
p = [-1, 0, 1, 0, 0, 0, 0, 3]
```

- Must store all zero coefficients, think about $1 + x^{100}$...
- Evaluating the polynomial at a given x value: $\sum_{i=0}^{N} c_i x^i$

```
def poly2(data, x):
    sum = 0
    for power in range(len(data)):
        sum += data[power]*x**power
    return sum
```

What is best for polynomials: lists or dictionaries?

- Dictionaries need only store the nonzero terms
- Dictionaries can easily handle negative powers, e.g., $\frac{1}{2}x^{-3} + 2x^4$

  ```
  p = {-3: 0.5, 4: 2}
  ```

- Lists need more book-keeping with negative powers:

  ```
  p = [0.5, 0, 0, 0, 0, 0, 0, 4]
  # p[i] corresponds to power i-3
  ```

- Dictionaries are much more suited for this task

Exercise

- Write a function named getBaseCounts2() which takes a string as input. The input string may contain letters other than A, C, T, and G. The function should return the counts of only A, C, T, and G in the form of a dictionary; it must ignore all letters other than A, C, T, and G. Test your function on a string such as 'ADLSTTLLD'.

Example: read file data into a dictionary

- Here is a data file:

```
Oslo:           21.8
London:         18.1
Berlin:         19
Paris:          23
Rome:           26
Helsinki:       17.8
```

- City names = keys, temperatures = values

```
infile = open('deg2.dat', 'r')
temps = {}                         # start with empty dict
for line in infile.readlines():
    city, temp = line.split()
    city = city[:-1]               # remove last char (:)
    temps[city]  = float(temp)
```

Reading file data into nested dictionaries

- Data file `table.dat` with measurements of four properties:

```
         A       B        C       D
1     11.7   0.035     2017    99.1
2      9.2   0.037     2019   101.2
3     12.2     no        no   105.2
4     10.1   0.031       no   102.1
5      9.1   0.033     2009   103.3
6      8.7   0.036     2015   101.9
```

- Create a dict `data[p][i]` (dict of dict) to hold measurement no. `i` of property `p` (`"A"`, `"B"`, etc.)
- Examine the first line: split it into words and initialize a dictionary with the property names as keys and empty dictionaries ({}) as values
- For each of the remaining lines: split line into words
- For each word after the first: if word is not "no", convert to float and store
- See the book for implementation details!

Comparing stock prices (part 1)

- Problem: we want to compare the stock prices of Microsoft, Sun, and Google over a long period
- finance.yahoo.com offers such data in files with tabular form

```
Date,Open,High,Low,Close,Volume,Adj Close
2008-06-02,28.24,29.57,27.11,28.35,79237800,28.35
2008-05-01,28.50,30.53,27.95,28.32,69931800,28.32
2008-04-01,28.83,32.10,27.93,28.52,69066000,28.42
2008-03-03,27.24,29.59,26.87,28.38,74958500,28.28
2008-02-01,31.06,33.25,27.02,27.20,122053500,27.10
...
```

- Columns are separated by comma
- First column is the date, the final is the price of interest
- We can compare Microsoft and Sun from e.g. 1988 and add in Google from e.g. 2005
- For comparison we should normalize prices: Microsoft and Sun start at 1, Google at the max Sun/Microsoft price in 2005

Comparing stock prices (part 1)

- Problem: we want to compare the stock prices of Microsoft, Sun, and Google over a long period
- finance.yahoo.com offers such data in files with tabular form

```
Date,Open,High,Low,Close,Volume,Adj Close
2008-06-02,28.24,29.57,27.11,28.35,79237800,28.35
2008-05-01,28.50,30.53,27.95,28.32,69931800,28.32
2008-04-01,28.83,32.10,27.93,28.52,69066000,28.42
2008-03-03,27.24,29.59,26.87,28.38,74958500,28.28
2008-02-01,31.06,33.25,27.02,27.20,122053500,27.10
...
```

- Columns are separated by comma
- First column is the date, the final is the price of interest
- We can compare Microsoft and Sun from e.g. 1988 and add in Google from e.g. 2005
- For comparison we should normalize prices: Microsoft and Sun start at 1, Google at the max Sun/Microsoft price in 2005

Comparing stock prices (part 1)

- Problem: we want to compare the stock prices of Microsoft, Sun, and Google over a long period
- finance.yahoo.com offers such data in files with tabular form

```
Date,Open,High,Low,Close,Volume,Adj Close
2008-06-02,28.24,29.57,27.11,28.35,79237800,28.35
2008-05-01,28.50,30.53,27.95,28.32,69931800,28.32
2008-04-01,28.83,32.10,27.93,28.52,69066000,28.42
2008-03-03,27.24,29.59,26.87,28.38,74958500,28.28
2008-02-01,31.06,33.25,27.02,27.20,122053500,27.10
...
```

- Columns are separated by comma
- First column is the date, the final is the price of interest
- We can compare Microsoft and Sun from e.g. 1988 and add in Google from e.g. 2005
- For comparison we should normalize prices: Microsoft and Sun start at 1, Google at the max Sun/Microsoft price in 2005

Comparing stock prices (part 1)

- Problem: we want to compare the stock prices of Microsoft, Sun, and Google over a long period
- finance.yahoo.com offers such data in files with tabular form

```
Date,Open,High,Low,Close,Volume,Adj Close
2008-06-02,28.24,29.57,27.11,28.35,79237800,28.35
2008-05-01,28.50,30.53,27.95,28.32,69931800,28.32
2008-04-01,28.83,32.10,27.93,28.52,69066000,28.42
2008-03-03,27.24,29.59,26.87,28.38,74958500,28.28
2008-02-01,31.06,33.25,27.02,27.20,122053500,27.10
...
```

- Columns are separated by comma
- First column is the date, the final is the price of interest
- We can compare Microsoft and Sun from e.g. 1988 and add in Google from e.g. 2005
- For comparison we should normalize prices: Microsoft and Sun start at 1, Google at the max Sun/Microsoft price in 2005

Comparing stock prices (part 1)

- Problem: we want to compare the stock prices of Microsoft, Sun, and Google over a long period
- finance.yahoo.com offers such data in files with tabular form

```
Date,Open,High,Low,Close,Volume,Adj Close
2008-06-02,28.24,29.57,27.11,28.35,79237800,28.35
2008-05-01,28.50,30.53,27.95,28.32,69931800,28.32
2008-04-01,28.83,32.10,27.93,28.52,69066000,28.42
2008-03-03,27.24,29.59,26.87,28.38,74958500,28.28
2008-02-01,31.06,33.25,27.02,27.20,122053500,27.10
...
```

- Columns are separated by comma
- First column is the date, the final is the price of interest
- We can compare Microsoft and Sun from e.g. 1988 and add in Google from e.g. 2005
- For comparison we should normalize prices: Microsoft and Sun start at 1, Google at the max Sun/Microsoft price in 2005

Comparing stock prices (part 1)

- Problem: we want to compare the stock prices of Microsoft, Sun, and Google over a long period
- finance.yahoo.com offers such data in files with tabular form

```
Date,Open,High,Low,Close,Volume,Adj Close
2008-06-02,28.24,29.57,27.11,28.35,79237800,28.35
2008-05-01,28.50,30.53,27.95,28.32,69931800,28.32
2008-04-01,28.83,32.10,27.93,28.52,69066000,28.42
2008-03-03,27.24,29.59,26.87,28.38,74958500,28.28
2008-02-01,31.06,33.25,27.02,27.20,122053500,27.10
...
```

- Columns are separated by comma
- First column is the date, the final is the price of interest
- We can compare Microsoft and Sun from e.g. 1988 and add in Google from e.g. 2005
- For comparison we should normalize prices: Microsoft and Sun start at 1, Google at the max Sun/Microsoft price in 2005

Comparing stock prices (part 1)

- Problem: we want to compare the stock prices of Microsoft, Sun, and Google over a long period
- finance.yahoo.com offers such data in files with tabular form

```
Date,Open,High,Low,Close,Volume,Adj Close
2008-06-02,28.24,29.57,27.11,28.35,79237800,28.35
2008-05-01,28.50,30.53,27.95,28.32,69931800,28.32
2008-04-01,28.83,32.10,27.93,28.52,69066000,28.42
2008-03-03,27.24,29.59,26.87,28.38,74958500,28.28
2008-02-01,31.06,33.25,27.02,27.20,122053500,27.10
...
```

- Columns are separated by comma
- First column is the date, the final is the price of interest
- We can compare Microsoft and Sun from e.g. 1988 and add in Google from e.g. 2005
- For comparison we should normalize prices: Microsoft and Sun start at 1, Google at the max Sun/Microsoft price in 2005

Comparing stock prices (part 2)

Algorithm for file reading:

- Skip first line, read line by line, split line wrt. colon, store first "word" in a list of dates, final "word" in a list of prices; collect lists in dictionaries with company names as keys; make a function so it is easy to repeat for the three data files

Algorithm for file plotting:

- Convert year-month-day time specifications in strings into coordinates along the x axis (use month indices for simplicity), Sun/Microsoft run 0,1,2,... while Google start at the Sun/Microsoft index corresponding to Jan 2005

See the book for all details. If you understand this example, you know and understand a lot!

Comparing stock prices (part 2)

Algorithm for file reading:

- Skip first line, read line by line, split line wrt. colon, store first "word" in a list of dates, final "word" in a list of prices; collect lists in dictionaries with company names as keys; make a function so it is easy to repeat for the three data files

Algorithm for file plotting:

- Convert year-month-day time specifications in strings into coordinates along the x axis (use month indices for simplicity), Sun/Microsoft run 0,1,2,... while Google start at the Sun/Microsoft index corresponding to Jan 2005

See the book for all details. If you understand this example, you know and understand a lot!

Comparing stock prices (part 2)

Algorithm for file reading:

- Skip first line, read line by line, split line wrt. colon, store first "word" in a list of dates, final "word" in a list of prices; collect lists in dictionaries with company names as keys; make a function so it is easy to repeat for the three data files

Algorithm for file plotting:

- Convert year-month-day time specifications in strings into coordinates along the x axis (use month indices for simplicity), Sun/Microsoft run 0,1,2,... while Google start at the Sun/Microsoft index corresponding to Jan 2005

See the book for all details. If you understand this example, you know and understand a lot!

Dictionary are Mutable Objects

Example:

```
x = {'a': 5}
y = x
x['a'] = 3
y = ?
```

```
x = {'a': [1,2,3]}
y = copy.copy(x)
x['a'] = [4,5,6]
y = ?
```

Dictionary with Default Value

```
from collections import defaultdict
def polynomial_coeff_default():
        #default value
        return 0.0
p2 = defaultdict(polynomial_coeff_default)
p2.update(p1) #p1 is an ordinary dictionary
```

Dictionary with Ordering

from collections import OrderedDict

data = OrderedDict()

data[‘Jan 2’] = 33

data[‘Jan 16’] = 0.1

data[‘Feb 2’] = 2

Does sorted(data) work in this example?

Exercise

Week 4, Cohort Session Problems, Question 9