Using Python for Scientific Computing
Session 2 - More Python and Working with Files and Databases

Felix Steffenhagen

University of Freiburg

May 3, 2011
1. More about functions

2. Modules - Organizing program structure

3. Object-oriented programming
   - Classes and Objects

4. Working with files

5. Working with CSV files

6. Accessing Databases

7. Regular Expressions
More about functions

- Python functions can be used and designed in various ways
- Named arguments for more intuitive function calls
- Default arguments
- Variable argument lists
Let’s have a look at the following function:

```python
def power(base, exponent):
    return base ** exponent
```

- When calling a function, parameters can be named
  - `power(base=2, exponent=10)` 1024
  - `power(exponent=10, base=2)` 1024
  - `power(2, exponent=10)` 1024

- Better code readability

- arbitrary¹ order of function parameters

¹Nearly. Named arguments always after unnamed arguments in function call.
Named arguments are especially useful with default arguments.

Default arguments can be left out in function call ⇒ Default value

def power(base, exponent=2, debug=False):
    if debug:
        print base, exponent
    return base ** exponent

Now we can call this function as in the following:

- power(10) 100
- power(10, 3 1000
- power(10, debug=True) 10, 2; 100
- power(debug=True, base=4) 4, 2; 16
In some cases it is useful if functions can accept an arbitrary number of (additional) arguments. E.g. computing the mean of values.

Idea: Collect additional parameters in a tuple

The parameter holding this tuple has to be marked by a "*"

```python
def mean(first, *args):
    n = 1.0 * len(args)
    return (first + sum(args)) / n
```

```python
mean(2.5)  # 2.5
mean(1, 2, 3, 4, 5)  # 7.5
```
Variables number of keyword arguments

- Functions can also accept an arbitrary number of keyword arguments. Very useful e.g. for passing options.
- Idea is the same as variable arguments list: Collect (additional) keyword arguments in a dictionary.
- The parameter holding this dictionary has to be marked by "**".

```python
def varargs(*args, **kwargs):
    print "args:", args
    print "kwargs:", kwargs

varargs(1, "one", x=1, y=2, z=3)
# args: (1, 'one')
# kwargs: {'x': 1, 'y': 2, 'z': 3}
```
When programs grow large, code is usually organized in conceptual units.

In Python, such conceptual units are called *Modules*.

Every Python file (*.py) is a module and vice versa.

To access external modules you use the `import` statement:

```python
import mymodule
from mymodule import X, Y
from mymodule import *
```
Example: myfuncts.py

For the definition of a module, we just write a normal .py file:

```python
myfuncts.py

my_pi = 3.14

def product(x, y, z):
    return x * y * z

def mean(first, *args):
    n = 1.0 * len(args)
    return (first + sum(args)) / n

print "loaded myfuncts.py"
```

This module can now be imported in other modules/programs.
After simple `import`, the defined structures must be accessed by:

```
<modulename>..<structure>
```
More ways of importing a module

- Module structures can be imported directly.
  Advantage: No "." notation necessary.

```python
import example2.py
from myfuncs import my_pi, product
print product(1, my_pi, 2)  # Output: 6.28
```

- You can also directly import all structures by using "*"

```python
import example3.py
from myfuncs import *
p = product(1, 2, 3)
print mean(my_pi, p, 2.0)  # Output: 3.71
```
A short Built in Safari

- **sys and os**: System-specific and parameters and functions
  - `sys.argv`: list of command line arguments
  - `sys.exit()`: Exit from Python

- **os**: Miscellaneous operating system interfaces

- **math/cmath**: Mathematical functions
  - `math`: power/logarithms, trigonometry, constants, ...
  - `cmath`: mathematical functions for complex numbers

- **re**: Regular Expression operations
  - More on that later

- **csv**: CSV File Reading and Writing
  - More on that later

- **optparse**: Powerful command line arguments and option parser

- **sqlite3**: Interface for SQLite databases

Felix Steffenhagen (Uni Freiburg)
Object-oriented programming

- Python is an object-oriented language
- OOP: Classes, Objects, Methods
- Organize code in conceptual units
- Reusability

```python
class Tree(object):
    def __init__(self, data, children=[]):
        self.data = data  # attribute data
        self.children = children  # attribute children

    def prettyprint(self, indent=' '):
        print(indent + self.data)
        for child in self.children:
            child.prettyprint(indent + '  ')
```
So how does it work?

- It defines a class `Tree` inherited from the class `object`.
- The class has a constructor taking the two parameters `data` and `children`.
- The class has a method `prettyprint`. 

```python
class Tree(object):
    def __init__(self, data, children=()):
        """The constructor method.""
        self.data = data  # attribute data
        self.children = children  # attribute children

    def prettyprint(self, indent=""):  
        """Prints the complete tree."""

        print indent + self.data

        for child in self.children:
            child.prettyprint(indent + "  ")
```
Anatomy of a class

```python
class MyClass(BaseClass):
    class_attr = value  # a class attribute
    def __init__(self, arg, ...):
        # class initialization
        self.attr = arg
    def method(self, args, ...):
        # do something
```

- Definition introduced by `class` followed by class name and one or more base classes (Use `object` for standard classes)
- First argument of methods is the `self` pointer, the reference to the calling object (Java or C++ uses `this`)
- The special `__init__` method is the constructor method and will be called whenever an instance of this class is created.
- Access to an attribute always using `self.attribute`
Creating and using Instances of a Class

family.py

```python
from tree import Tree

nephews = (Tree("Tick"), Tree("Trick"), Tree("Track"))
donald, daisy = Tree("Donald", nephews), Tree("Daisy")
dagobert = Tree("Dagobert", (donald, daisy))
dagobert.prettyprint()

# Dagobert
#   Donald
#     Tick
#     Trick
#     Track
#   Daisy
```
Class attributes vs. Instance attributes

- Attributes can be set dynamically at any time
- Class attributes are the same for all instances of the class.
- Instance attributes are specific for a single instance.

```
>>> from tree import Tree
>>> donald, daisy = Tree("Donald"), Tree("Daisy")
>>> donald.age = 35
>>> daisy.age = 33
>>> print donald.age, daisy.age
35 33
>>> Tree.count = 2
>>> print donald.count, daisy.count
2 2
```
Further OOP topics

This should be all for now to be able to work with classes.

There are much much more topics about object-oriented programming in Python:

- Static methods
- Magic methods (operator overloading)
- Multiple Inheritance
- Computed attributes (*properties*)

- Now we know enough about the basic Python interna and syntax
- Let’s now have a look about how to access stored data (files, DBs)
Files are one way of communicating with the outside world.

Read/Write data from/to files

In Python the function `open` creates a file object for reading and writing

File objects are iterable, i.e. we can use the for loop for iterating over the lines

```python
linecount.py

f = open("filename.txt", "r")  # open file for reading
count = 0
for line in f:
    count +=1
print "Counted %s lines" % count
f.close()  # close file object
```
The function `open` in detail

- `open(filename[, mode[, bufsize]])`:
  - Opens the file `filename` and returns a file object for I/O
  - `mode`, determines how the file will be opened
    - "r", "rb": open for reading; "rb" for binary files
    - "w", "wb": open for writing; "wb" for binary files
      - Careful: Existing files will be overwritten
    - "r+", "r+b": open for reading and writing
    - "a", "ab": open for appending; writes at the end of a file
  - `bufsize`, determines file buffering
    - negative value (default): Let the OS decide
    - 0: no buffering
    - 1: single line buffering
    - x > 1: buffer ca. x bytes

Most common calls of `open`

```python
>>> f_in = open(infile, "r")  # read from a file
>>> f_out = open(outfile, "w")  # write a file
```
Simplest way to read from files is by iteration:

```python
for line in f:
    # When reading a file you’ll mostly want this.
```

*f.read([size])*: reads size bytes from the file and returns them as string. If no size specified, the whole file is returned in a single string.

```python
f.readline()  # reads a single line and returns it as a string
f.readlines()  # Repeatedly calls readline and returns a list of strings
```
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:

```python
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>
```
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!
We must convert strings to numbers before computing:

```python
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:

```python
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:

```python
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

```python
>>> infile = open('data1.txt', 'r')
>>> fstr = infile.read()  # read file into a string
>>> fstr
'21.8
18.1
19
23
26
17.8
'
>>> 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
', '18.1
', '19
', '23
', '26
', '17.8
']
>>> infile.close(); infile = open('data1.txt', 'r')
>>> for line in infile: print line,
... 21.8
 18.1
 19
 23
 26
 17.8
```
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

```python
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(’;’)`
The complete program:

```python
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')
```
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.
Writing to files

- `f.write(string)`: Writes a string to the file object `f`. Newlines ("\n") are not automatically inserted.

- `f.writelines(sequence)`: Writes a sequence of strings to `f`. Despite the function name, newlines are not automatically inserted.

- `f.flush()`: File I/O is usually buffered and not instantly written to the file. `flush` clears the buffer and writes the data to the file.

- One can also use the `print` function to directly print into files: `print >> f, "This is a line"`
Example: file copy

```python
filecopy.py

f_src = open("original.txt", "r")
f_dst = open("copy.txt", "w")

for line in f_src:
    f_dst.write(line)
```

Here we don’t need to write newline characters into the copy because they are already in the original line.

```
original.txt
  first line
  second line
  third line

copy.txt
  first line
  second line
  third line
```
Working with CSV files

- Often data come in tabular shape (e.g. Excel sheet)
- CSV (Comma Separated Value) is a simple and platform independent file format
- Most programs can deal with CSV files
- Beside commas, other separators (e.g., tabs, semicolon)

<table>
<thead>
<tr>
<th>id, x, y, z, alpha, beta</th>
</tr>
</thead>
<tbody>
<tr>
<td>AT42, 1.0, 5.2, 4.2, 2.3, 6.6</td>
</tr>
<tr>
<td>AT42, 5.0, 6.9, 2.8, 2.7, 9.1</td>
</tr>
<tr>
<td>AT22, 4.0, 5.9, 8.1, 3.7, 4.7</td>
</tr>
<tr>
<td>AT22, 5.0, 6.9, 3.8, 2.7, 4.1</td>
</tr>
</tbody>
</table>

...
The builtin module **csv**

- **csv** provides classes for easy reading/writing of CSV files.
- No need for manual string splitting
- **reader** and **writer** classes for reading/writing sequences of data
- **DictReader** and **DictWriter** process data in dictionary form

### Short CSV reader example

```python
import csv
f = open("data.csv")  # open file for reading
dat = csv.reader(f, delimiter="","
for row in data:
    print(row)  # each row is a list of strings!

# Output
# ['id', 'x', 'y', 'z', 'alpha', 'beta']
# ['AT42', '1.0', '5.2', '4.2', '2.3', '6.6']
```
The CSV reader/writer classes

- `csv.reader(csvfile[, dialect=excel][, fmtparam])`:
  Creates a reader object for iterating over `csvfile`. Optional `dialect` specifies CSV dialect\(^2\). Additional format keyword parameters can be given to specify CSV format.

- `csv.writer(csvfile[, dialect=excel][, fmtparam])`:
  Analogous to above, but creates a writer object.

- **Important format parameters:**
  - `delimiter`: string specifying value separator
  - `quotechar`: string specifying how text values are quoted

- **Reading and Writing:**
  - Reading mostly done by looping over reader object
  - Writing a single row (list of values): `csvwriter.writerow(row)`
  - Writing several rows (list of rows): `csvwriter.writerows(rows)`

\(^2\) Mostly not necessary.
Copy the last three columns of a CSV file into a new CSV file:

csv_example.py

```python
import csv
f_in = open("data.csv")        # open input file

csv_r = csv.reader(open(f_in)  # create csv reader
csv_w = csv.writer(open("newdata.csv", "w"))

for row in csv_r:
    # process row; write last three columns
    csv_w.writerow(row[-3:])
    csv_w.writerow([3,2,1])
```

Felix Steffenhagen (Uni Freiburg)
DictReader and DictWriter

- Mostly CSV files include row headers to name the columns
- **Standard reader and writer** only work with lists
- More intuitive to access elements by their column names
- Idea: Read rows into dictionaries regarding the CSV column names.

DictReader usage example

```python
from csv import DictReader
f = open("data.csv")     # open file for reading
dat = DictReader(f, delimiter="","")
for row in dat:
    print(row)            # now each row is dict!

# Output
# {‘id’: ’AT42’, ’x’: ’1.0’, ’y’: ’5.2’, ... }
```
**DictReader and DictWriter classes**

- `csv.DictReader(csvfile[, fieldnames=None[, *args [, **kwargs]]]]):`
  - Creates a DictReader object for iterating over `csvfile`.
    - `fieldnames`: string list specifying dict keywords.
      - If omitted, entries of first line will be used.
    - additional parameter are passed to underlying reader class

- `csv.DictWriter(csvfile, fieldnames[, *args [, **kwargs]]]):`
  - Analogous to above, but creates a writer object. Note: You have to specify a list of strings, containing the CSV field names.
  - Functions for reading and writing the same as for standard reader/writer objects
Using the DictReader and Writer

```python
from csv import DictReader, DictWriter
fields = ["id", "length", "width", "height"]
dr = DictReader(open("data.csv"))
dw = DictWriter(open("newdata.csv", "w"), fields)
    # insert header line
    dw.writerow(dict(zip(fields, fields)))
for row in dr:
    new_row = {}
        # new empty dict
    new_row["id"] = row["id"]
    new_row["length"] = row["x"]
    new_row["width"] = row["y"]
    new_row["height"] = row["z"]
    dw.writerow(new_row)    # write row
```

Note: You have to manually insert a header line to the new file!
Multiple iteration over files

- Careful when iterating over files several times!
- `csv` Reader objects only scan once through the file.
- For further iteration you have to reset the file cursor

```python
from csv import DictReader
f = open("data.csv")  # open file for reading
dr = DictReader(f, delimiter="","")
print(len([row for row in dr]))  # print row count
print(len([row for row in dr]))  # print again
f.seek(0)  # reset file cursor
print(len([row for row in dr]))  # print again
# Output: 4 0 4
```
Example: read pairs of numbers \((x,y)\) from a file

**Sample file:**

\[
\begin{align*}
(1.3,0) & \quad (-1,2) & \quad (3,-1.5) \\
(0,1) & \quad (1,0) & \quad (1,1) \\
(0,-0.01) & \quad (10.5,-1) & \quad (2.5,-2.5)
\end{align*}
\]

**Method:** read line by line, for each line: split line into words, for each word: split off the parenthesis and the split the rest wrt comma into two numbers.
The code for reading pairs

```python
text = 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)]
Plain file storage works fine if processing of all data is necessary.

Databases allow storing large amounts of structured data and are good for processing partial data sets.

Python builtin library supports SQLite (**sqlite3**) and Free modules for PostgreSQL (**psycopg2**) and MySQL (**MySQLdb**) Similar access workflow for the different DB systems (DB-API 2.0)

\[a^{http://www.initd.org/psycopg/}\]
\[b^{http://mysql-python.sourceforge.net/}\]
import dbapi

# create connection to DB
conn = dbapi.connect( db_params )

# create cursor for DB communication
cursor = conn.cursor()

# execute SQL statement
cursor.execute(sql_stmt)

# fetch results or commit changes
cursor.fetchall(), cursor.commit()

# close db connection
conn.close()
SQLite

- SQLite is a C library that provides lightweight, disk-based databases.
- No server configuration overhead, it just works.
- Portable databases in a single file.
- SQLite allows multiple applications to access the same DB.
- Disadvantages:
  - Slower than big DBMS, but mostly not very much.
  - Lacking support for scalability.
Creating or using a database

- SQLite comes with a small command-line tool for DB management
- We can also create new DBs by creating a new connection to a not existing DB.

```python
import sqlite3

# create or open database
con = sqlite3.connect("mydata.db")
cursor = con.cursor()
```
Now use cursor (CURrent Set Of Records) to execute SQL statements

Let’s create a new table and insert some data:

db_example.py (cntd.)

cursor.execute("""CREATE TABLE expdata (id TEXT, x REAL, y REAL, z REAL)""")

cursor.execute("""INSERT INTO expdata (id, x, y, z) VALUES ("probe1", 10, 7, 5)""")

values = ["probe2", 15], ["probe3", 22], ["probe4", 11]
cursor.executemany("""INSERT INTO expdata (id, x, y, z) VALUES (?, ?, 30, 9)"", ids)

con.commit()
SELECT queries return rows of data.

```python
# db_example.py (cntd.)
cursor.execute("""SELECT * FROM expdata"""")
for row in cursor:
    print row

# (u'probe1', 10.0, 7.0, 5.0)
# (u'probe2', 15.0, 30.0, 9.0)
# (u'probe3', 22.0, 30.0, 9.0)
# (u'probe4', 11.0, 30.0, 9.0)

'u' marked strings are unicode strings
Query results can be fetched individually or all at once.

- `cursor.fetchone()`: Fetches next row or `None` if no more rows available
- `cursor.fetchall()`: Fetches all (remaining) rows, returning a list

```python
cursor.execute("""SELECT * FROM expdata""")
first = cursor.fetchone()
rest = cursor.fetchall()
for row in rest:
    print row  # (u'probe2', 15.0, 30.0, 9.0)
    # (u'probe3', 22.0, 30.0, 9.0)
print first  # (u'probe4', 11.0, 30.0, 9.0)
    # (u'probe1', 10.0, 7.0, 5.0)
```
Three important methods for executing SQL statements

- `cursor.execute(sql, params)`: Executes a single SQL statement. Statements can be parametrized by placeholders that are replaced by `params`.

- `cursor.executemany(sql, param_seq)`: Executes as many statements as are in `param_seq`. Useful to send the same statement with different parameters.

- `cursor.executescript(sql_script)`: Executes multiple SQL statements at once. Useful for executing external SQL scripts.
Often, SQL statements are created dynamically
Since statements are only strings one could assemble them by using %s interpolation.

NEVER DO THIS! – INSECURE!

```python
ame = "probe2"
cursor.execute("SELECT * FROM expdata
    WHERE id = '%s'" % name)
```

This might lead to SQL injection!!!
Instead, use parameter substitution of the `execute` command
(See next slide)
The correct way to do it is by using parameter substitution. Put `?` placeholder wherever you want to use a parameter value and provide a parameter tuple to `execute` method.

**DO THIS INSTEAD**

```python
cursor.execute("""SELECT * FROM expdata
    WHERE id = ?""", name)
```
Parametrized `execute` examples

db_example.py (cntd.)

```python
values = [ ('probe5', 3.1, 1.4, 4.3),
          ('probe6', 3.9, 8.7, 7.2)]

# using for loop to send separate inserts
for value in values:
    cursor.execute("""INSERT INTO expdata
                     VALUES (?,?,?,?)""", value)

# The same without a for loop
cursor.executemany("""INSERT INTO expdata
                     VALUES (?,?,?,?)""", values)
```
Executing SQL scripts

- Scripts are useful for storing often used statements, e.g. for database creation.

**createdb.sql – note the semicolon at the end of each statement**

```sql
CREATE TABLE expdata
    (id TEXT, x REAL, y REAL, z REAL);
CREATE TABLE parameters
    (id TEXT, alpha REAL, beta REAL, gamma REAL);
```

**Executing a sql script from a file**

```python
script = open("createdb.sql").readlines()
cursor.execute(script)
```
Regular Expressions - The `re` module

- Small specialized programming language.
- Describe patterns for sets of strings (e.g. email addresses).
  
  **example:** `spam\d+[a-z]{2,3}`

- The builtin module `re` provides Perl-style (most widely used) regex patterns
- Compiled into state-machines on which you apply a string as input.
- What to do with it?
  
  - Matching string patterns (e.g. user input)
  - Finding strings according to a pattern

Shakespeare

```python
>>> import re
>>> p = re.compile(r"bb|[^b]{2}\")
>>> match = p.match("aac")
>>> match.group()
'aa'
```
Basic Structure of Regular Expressions

- **Reserved meta symbols:** ., ^, $, +, ?, {, }, (, ), [, ], \
- Any character or complete words.
- **Character classes:** [a-c], [a-zA-Z], [ace]:
- **Repetitions:** \ab* (zero or more b’s), \ab+ (at least 1 b), \ab? (zero or one b), \ab{2, 5} (2-5 b’s)

**Backslash Characters**
- \d: digit characters ([0-9])
- \D: non-digit characters (^[0-9])
- \s: whitespace characters (\[\t\n\r\f\v\])
- \S: non-whitespace characters (^[\t\n\r\f\v\])
- \w: alphanumeric characters ([a-zA-Z0-9_])
- \W: non-alphanumeric characters ([^a-zA-Z0-9_])

. stands für any character.
19 Jan 2038 at 3:14:07 AM

(End of the word according to Unix—$2^{32}$ seconds after January 1, 1970)

```python
import re
text = "19 Jan 2038 at 3:14:07 AM"
time = r"\d{1,2}:\d{2}:\d{2}\s[A|P]M"
print re.findall(time, text)
# output: ['3:14:07 AM']
```
import re

text = "19 Jan 2038 at 3:14:07 AM"

time = r"\d{1,2}:\d{2}:\d{2}\s[A|P]M"

print re.findall(time, text)
# output: ['3:14:07 AM']

year = r"\d{4}"

print re.findall(year, text)
# output: ['2038']
import re

text = "19 Jan 2038 at 3:14:07 AM"

time = r"\d{1,2}:\d{2}:\d{2}\s[A|P]M"
print re.findall(time, text)
# output: ['3:14:07 AM']

year = r"\d{4}" 
print re.findall(year, text)
# output: ['2038']

daymon = r"\d{1,2}\s[A-Z][a-z]\{2}" 

date = daymon + r"\s" + year 
print re.findall(date, text)
# output: ['19 Jan 2038']

19 Jan 2038 at 3:14:07 AM

(End of the word according to Unix–2^32 seconds after January 1, 1970)
The Backslash Plague

• Regular expressions often contain Backslashes.
• Backslashes have special meaning in normal Python strings. Must be escaped by another Backslash e.g. "\\n"
• Raw strings:
  • strings with leading ’r’: e.g. r"\n stays \n here"
  • In raw strings, Backslashes are kept untouched
• Recommendation: Use raw strings when working with regex patterns.
Matching Strings

- `p.match(string)`: Tries to match pattern `p` at the beginning of `string`. Returns either Match object or `None` if no match.
- `match.group()`: Returns the first matching string group.

```python
import re
p = re.compile(r"FR-\d*[A-Z]2")
input = "FR-123IYhltz"
m = p.match(input)
print m.group()
# output: 'FR-123IY'
```

Note: `string` must not be completely equal to pattern. Beginning is sufficient for match.
Finding Strings

- `p.findall(string)`:
  Returns a list of all non-overlapping matches of pattern in string.

```python
import re
input = """The thirty-three thieves thought that they thrilled the throne throughout Thursday."""
p = re.compile(r"thr\w+"")
print p.findall(input)
# output: ['three', 'thrilled', 'throne', 'throughout']
```
There are some more useful metacharacters:

- \A: matches the beginning of the string
- \b: matches word boundary (a whitespace or non-alphanumeric character)

### match exactly two words

```python
import re
p = re.compile(r"\A\w+s\w+\Z")
p.match("Monty Pythons Flying Circus")
# None
print p.match("Monty Python")
# output: ['three', 'thrilled', 'throne', 'throughout']
```
Further Topics

- Groupings
- Lookahed Assertions
- Non-Capturing Groups
- RegEx on multiline texts
- Greedy vs. Non-Greedy Search

See Andrew Kuchlings *Python Regular Expression HOWTO*. 