

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

- Python functions can be used and designed in various ways
- Named arguments for more intuitive function calls
- Default arguments
- Variable argument lists

Named arguments

Let's have a look at the following function:

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

- When calling a function, parameters can be named

• <code>power(base=2, exponent=10)</code>	1024
• <code>power(exponent=10, base=2)</code>	1024
• <code>power(2, exponent=10)</code>	1024

- Better code readability
- arbitrary¹ order of function parameters

¹Nearly. Named arguments always after unnamed arguments in function call.

Default arguments

- Named arguments are especially useful with *default arguments*.
- Default arguments can be left out in function call $\Rightarrow$ 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 |

Variable argument lists

- 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 "*"

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

```
mean(2.5)                # 2.5
```

```
mean(1, 2, 3, 4, 5)      # 7.5
```

Variable 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 "**"

```
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}
```

Organizing program code in modules

- 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:

```
import mymodule  
from mymodule import X, Y  
from mymodule import *
```

Example: myfuncs.py

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

myfuncs.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 myfuncs.py"
```

This module can now be imported in other modules/programs.

Simple import

```
import_example1.py
```

```
print("Start")                                # Output: Start

import myfuncs                                # Output: loaded myfuncs

print(myfuncs.product(2, 3, 4))               # Output: 24
print(myfuncs.mean(10, 20, 30))              # Output: 20.0
print(myfuncs.my_pi)                         # Output: 3.14
```

- 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.

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 "*"

import_example3.py

```
from myfuncs import *
p = product(1, 2, 3)
print mean(my_pi, p, 2.0)           # Output: 3.71
```

A short Builtin Safari

- `sys` and `os`: System-specific 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

Object-oriented programming

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

tree.py

```
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 + "  ")
```

Example

tree.py

```
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 + "  ")
```

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`.

Anatomy of a class

class_template.py

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

```
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.

Attributes

```
>>> 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
```

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)

Working with files

- 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

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`

```
>>> f_in = open(infile, "r")           # read from a file
>>> f_out = open(outfile, "w")         # write a file
```

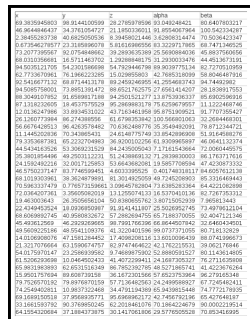
Reading from files

- Simplest way to read from files is by iteration:

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



x	y	z	alpha	beta
88.3835945803	98.9144100599	28.2785978936	83.0482484821	80.6407803217
46.8648464437	24.3761079477	21.1850126011	81.9954847864	100.5427341987
2.98455283738	40.6825050536	8.39458021446	3.628086314474	70.5036423347
0.67546278577	22.3185866078	9.81616889556	83.3229717885	68.7471348525
73.2077365567	82.0765484862	59.283935389	25.5680640336	45.8537560856
68.0320366681	16.5711483702	1.29288948175	31.2930033476	44.4513673191
84.9256121705	54.2301586668	54.792446738	89.8039776134	82.7270510569
82.7753870861	76.1866212395	15.028955803	42.7608318589	58.8064479318
82.5416677132	68.8714413178	89.2459246955	41.2954683743	84.74462982
84.5085758001	73.8851391472	88.6521762575	27.6561414207	28.1889179553
86.2646027852	91.6969817488	84.295121177	23.8793396337	65.6802586936
87.1318228005	18.4523757529	95.2869883178	75.6259679957	11.1222468746
12.0136247986	33.8934521022	43.7163461958	95.8791909521	91.7707355427
26.1240772884	88.27438956	41.6786353842	100.566801063	32.2684469301
56.6676428513	96.4263578482	70.6362488776	35.9549482091	78.8712344721
11.1445202836	70.343885431	24.6140175749	33.934286938	53.1645688176
78.3358877861	65.232784883	96.820031256	61.8795985897	46.0641123274
44.5434143526	53.3069231529	84.2435059543	7.17615143664	72.0060469575
75.3803185446	49.2923112231	31.2438849132	71.2639820053	61.7163717616
24.1592482216	32.0017159583	53.6843882081	10.5857709594	47.4230873332
46.5796237147	83.7746598451	4.603339525	0.401748318117	84.6057631218
88.1018300961	38.3624878891	83.3014250589	48.7345258963	95.316488443
70.5983337479	0.776573159861	1.09645762804	73.6385283864	64.4221268288
73.036207361	3.35605962019	1.31259074123	16.5370410136	82.7267353332
19.4630018443	28.3565656104	60.8308855762	3.80710552539	7.8656112441
22.4349452524	18.0293650387	91.8141411807	25.5526952745	73.4978932104
68.808892745	40.958082672	57.2862684755	65.7189370055	80.4047121146
45.403612569	48.2832928689	38.7891766386	66.8648027842	32.6404343001
48.5608225186	48.5541108376	41.3220401586	99.073771055	80.71812829
14.0169808976	47.1581284452	17.4095208136	1.8501008439	88.0741939673
21.3217076664	63.1890817477	62.8747474642	42.1782219531	38.0621768126
54.0115970147	23.2586939562	9.74868957502	52.880591527	80.143614805
61.5206203688	10.044502423	41.4872299411	24.1687235237	76.7171259508
95.9811963891	82.6515131469	86.7852180785	65.571865741	41.4222847624
31.9501757684	89.608739158	38.1872101566	57.6523753864	96.279455348
79.7520570192	79.8976870159	67.7136482563	24.2489589827	67.725484811
14.2548462621	10.8893752468	34.4718134389	64.389554468	74.7722789125
89.16819150518	27.958825771	95.65689623172	42.7466792136	65.62764817
31.168159792	30.3769850245	82.2018461076	70.1864224679	90.0002219514
84.1594320684	37.1884477875	30.1427661806	20.5767050538	70.853456939

Data file

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:
    line = infile.readline()
    if not line:
        break
    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

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

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
```

filecopy.py
⇒

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 commata, other separators (e.g., tabs, semicolon)

data.csv

```
id,x,y,z,alpha,beta
AT42,1.0,5.2,4.2,2.3,6.6
AT42,5.0,6.9,2.8,2.7,9.1
AT22,4.0,5.9,8.1,3.7,4.7
AT22,5.0,6.9,3.8,2.7,4.1
...
```

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

```
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 dialect². 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)`

²Mostly not necessary.

Example

Copy the last three columns of a CSV file into a new CSV file:

csv_example.py

```
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])
```

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

```
from csv import DictReader
f    = open("data.csv")           # open file for reading
dat = DictReader(f, delimiter=",")

for row in data:
    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

```
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

readtwice.py

```
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:

(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 parenthesis and then 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)]
```

- 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`)
- Free modules for PostgreSQL (`psycopg2a`) and MySQL (`MySQLdbb`)
- Similar access workflow for the different DB systems (DB-API 2.0)



^a<http://www.initd.org/psycopg/>

^b<http://mysql-python.sourceforge.net/>

Accessing Databases - A general template

db_template.py

```
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 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.

db_example.py

```
import sqlite3
# create or open database
con = sqlite3.connect("mydata.db")
cursor = con.cursor()
```

Creating new table and insert data

- Now use cursor (*CUR*rent 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()
```

Fetching Data

- SELECT queries return rows of data.

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

Fetching Data cntd.

- 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

db_example.py (cntd.)

```
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.

Parametric SQL Statements

- Often, SQL statements are created dynamically
- Since statements are only strings one could assemble them by using `%s` interpolation.

NEVER DO THIS! – INSECURE!

```
name = "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

```
name = "probe2"  
cursor.execute("""SELECT * FROM expdata  
                WHERE id = ?""", name)
```

db_example.py (cntd.)

```
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

```
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

```
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

```
>>> 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.

Examples

19 Jan 2038 at **3:14:07 AM**

(End of the word according to Unix-2³² seconds after January 1, 1970)

```
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']
```

Examples

19 Jan **2038** at 3:14:07 AM

(End of the word according to Unix-2³² seconds after January 1, 1970)

```
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']
```

19 Jan 2038 at 3:14:07 AM

(End of the word according to Unix-2³² seconds after January 1, 1970)

```
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']
```

The Backslash Plague

- Regular expressions often contain Backslashes.
- Backslashes have special meaning in normal Python strings. Must be *escape'd* 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.

```
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`.

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

words beginning with 'thr'

```
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']
```

More Metacharacters

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

```
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']
```

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

See Andrew Kuchlings *Python Regular Expression HOWTO*.