PYTHON
Day 4

Day 4 - Agenda

- Database connection
- Regular Expression
- Packages
- Python GUI

Database access

- The Python standard for database interfaces is the Python DB-API. Most Python database interfaces adhere to this standard.

- Python Database API supports a wide range of database servers:
  - GadFly
  - mSQL
  - MySQL
  - PostgreSQL
  - Microsoft SQL Server 2000
  - Informix
  - Interbase
  - Oracle
  - Sybase

- The DB API provides a minimal standard for working with databases using Python structures and syntax wherever possible. This API includes the following:
  - Importing the API module.
  - Acquiring a connection with the database.
  - Issuing SQL statements and stored procedures.
  - Closing the connection

Database access

- **What is MySQLdb?**
- MySQLdb is an interface for connecting to a MySQL database server from Python. It implements the Python Database API v2.0 and is built on top of the MySQL C API.

```
import MySQLdb
# Open database connection
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# prepare a cursor object using cursor() method
cursor = db.cursor()

# execute SQL query using execute() method.
cursor.execute("SELECT VERSION()")

# Fetch a single row using fetchone() method.
data = cursor.fetchone()

print "Database version : %s " % data # disconnect from server
db.close()
```

Regular Expressions

- **Background**
  - Regular expressions(called REs, or regexes, or regex patterns) are patterns that specify a matching rule.
  - Generally contain a mix of text and special characters
  - Example

foo.*	# Matches any string starting with foo
\d*	# Match any number decimal digits
[a-zA-Z]+	# Match a sequence of one or more letters

- **The re module**
  - Provides regular expression pattern matching and replacement.

Regular Expressions

- **Regular expression pattern rules**

text	Match literal text
.	Match any character except newline
^	Match the start of a string
$	Match the end of a string
*	Match 0 or more repetitions
+	Match 1 or more repetitions
?	Match 0 or 1 repetitions
*?	Match 0 or more, few as possible
+?	Match 1 or more, few as possible
{m,n}	Match m to n repetitions a{2,} – aa,aaa,aaaa,aaaa
{m,n}?	Match m to n repetitions, few as possible
[...]	Match a set of characters [a-d]*- abc, add, cd,
[^...]	Match characters not in set [^a-z] –
A | B	Match A or B
(...)	Match regex in parenthesis as a group

Regular Expressions

- **Special characters**

\number	Matches text matched by previous group
\A	Matches start of string
\b	Matches empty string at beginning or end of word
\B	Matches empty string not at begin or end of word
\d	Matches any decimal digit [0-9]
\D	Matches any non-digit
\s	Matches any whitespace
\S	Matches any non-whitespace
\w	Matches any alphanumeric character
\W	Matches characters not in \w
\Z	Match at end of string.
\\	Literal backslash

The re Module

- The re module provides an interface to the regular expression engine, allowing you to compile REs into objects and then perform matches with them.

- Regular expressions are compiled into pattern objects, which have methods for various operations such as searching for pattern matches or performing string substitutions.

```
>>> import re
>>> p = re.compile('ab*')
>>> p
<_sre.SRE_Pattern object at 0x...>
```

- re.compile() also accepts an optional flags argument, used to enable various special features and syntax variations.

- **Example:**

```
>>> p = re.compile('ab*', re.IGNORECASE)
```

The re Module

- The RE is passed to re.compile() as a string. REs are handled as strings because regular expressions aren’t part of the core Python language, and no special syntax was created for expressing them.

- There are applications that don’t need REs at all, so there’s no need to bloat the language specification by including them.)

- Instead, the re module is simply a C extension module included with Python, just like the socket or zlib modules.

Problem with Backslash

- Let’s say you want to write a RE that matches the string \section
- Next, you must escape any backslashes and other metacharacters by preceding them with a backslash, resulting in the string \\section.
- The resulting string that must be passed to re.compile() must be \\section.
- However, to express this as a Python string literal, both backslashes must be escaped again.

Characters	**Stage**
\section	Text string to be matched
\\section	Escaped backslash for re.compile()
"\\\\section"	Escaped backslashes for a string literal

The re Module

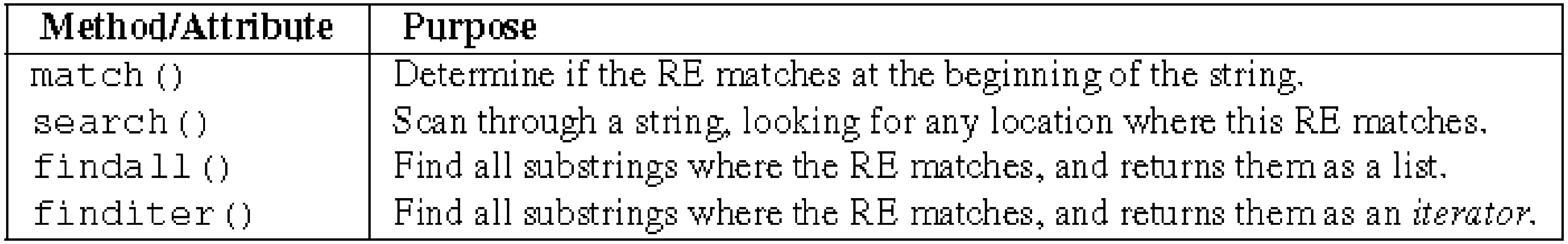
- **Raw strings**
  - Because of backslashes and special characters, raw strings are used.
  - Raw strings don’t interpret backslash as an escape code

  i.e., backslashes are not handled in any special way in a string literal prefixed with ’r’, so r"\n" is a two-character string containing ’\’ and ’n’, while "\n" is a one-character string containing a newline.

  - expr = r’(\d+)\.(\d*)’ # Matches numbers like 3.4772

Regular String	Raw string
"ab*"	r"ab*"
"\\\\section"	r"\\section"
"\\w+\\s+\\1"	r"\w+\s+\1"

Performing Matches



Method/Attribute	Purpose
`match()`	Determine if the RE matches at the beginning of the string.
`search()`	Scan through a string, looking for any location where this RE matches.
`findall()`	Find all substrings where the RE matches, and returns them as a list.
`finditer()`	Find all substrings where the RE matches, and returns them as an *iterator*.

The re Module

- **Regular Expression Objects**
  - Objects created by re.compile() have these methods
    - r.search(s [,pos [,endpos]]) # Search for a match
    - r.match(s [,pos [,endpos]]) # Check string for match
    - r.split(s) # Split on a regex match
    - r.findall(s) # Find all matches
    - r.sub(repl,s) # Replace all matches with repl
  - When a match is found a ’MatchObject’ object is returned.
  - This contains information about where the match occurred.
  - Also contains group information.

- **Notes**
  - The search method looks for a match anywhere in a string.
  - The match method looks for a match starting with the first character.
  - The pos and endpos parameters specify starting and ending positions for the search/match.

The re Module

```
>>> import re
>>> p = re.compile('[a-z]+')
>>> p.match("")
>>> print(p.match(""))
        None
>>> m = p.match('tempo123')
```

- match object instances also have several methods and attributes; the most important ones are:

Method/Attribute	Purpose
group()	Return the string matched by the RE
start()	Return the starting position of the match
end()	Return the ending position of the match
span()	Return a tuple containing the (start, end) positions of the match

```
>>> m.group()
'tempo'
>>> m.start(), m.end()
(0, 5)
>>> m.span()
(0, 5)
```

The re Module

```
>>> print(p.match(’::: message’))  #match() checks if the RE matches at the start of a string
None
>>> m = p.search(’::: message’); print(m)
<_sre.SRE_Match object at 0x...>
>>> m.group()
’message’
>>> m.span()
(4, 11)


>>> p = re.compile(’\d+’)
>>> p.findall(’12 drummers drumming, 11 pipers piping, 10 lords a-leaping’)
[’12’, ’11’, ’10’]

>>> iterator = p.finditer(’12 drummers drumming, 11 ... 10 ...’)
>>> iterator
<callable_iterator object at 0x...>
>>> for match in iterator:
... print(match.span())
...
(0, 2)
(22, 24)
(29, 31)
```

Module-Level Functions

- You don’t have to create a pattern object and call its methods; the re module also provides top-level functions called match(), search(), findall(), sub(), and so forth.

- These functions take the same arguments as the corresponding pattern method, with the RE string added as the first argument, and still return either None or a match object instance.

```
>>> print(re.match(r’From\s+’, ’Fromage amk’))
None
>>> re.match(r’From\s+’, ’From amk Thu May 14 19:12:10 1998’)
<_sre.SRE_Match object at 0x...>
```

Metacharacters

- **|** Alternation, or the “or” operator.
- If A and B are regular expressions, A|B will match any string that matches either A or B.
- (Crow|Servo) will match either Crow or Servo, not Cro, a ’w’ or an ’S’, and ervo.
- To match a literal ’|’, use \|, or enclose it inside a character class, as in [|].

Metacharacters

- **^** Matches at the beginning of lines.

- Unless the MULTILINE flag has been set, this will only match at the beginning of the string. In MULTILINE mode, this also matches immediately after each newline within the string.
- For example, if you wish to match the word From only at the beginning of a line, the RE to use is ^From.

```
>>> print(re.search(’^From’, ’From Here to Eternity’))
<_sre.SRE_Match object at 0x...>
>>> print(re.search(’^From’, ’Reciting From Memory’))
None
```

Metacharacters

- **\b** Word boundary.
- This is a zero-width assertion that matches only at the beginning or end of a word.
- A word is defined as a sequence of alphanumeric characters, so the end of a word is indicated by whitespace or a non-alphanumeric character.

```
>>> p = re.compile(r’\bclass\b’)
>>> print(p.search(’no class at all’))
<_sre.SRE_Match object at 0x...>
>>> print(p.search(’the declassified algorithm’))
None
>>> print(p.search(’one subclass is’))
None
```

Grouping

- Groups are marked by the ’(’, ’)’ metacharacters. ’(’ and ’)’ have much the same meaning as they do in mathematical expressions; they group together the expressions contained inside them, and you can repeat the contents of a group with a repeating qualifier, such as *, +, ?, or {m,n}.

- For example, (ab)* will match zero or more repetitions of ab.

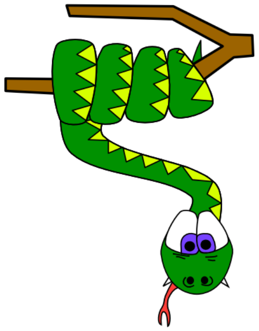
```
>>> p = re.compile(’(ab)*’)
>>> print(p.match(’ababababab’).span())
(0, 10)

>>> p = re.compile(’(a)b’)
>>> m = p.match(’ab’)
>>> m.group()
’ab’
>>> m.group(0)
’ab’
```

Grouping

- Subgroups are numbered from left to right, from 1 upward. Groups can be nested; to determine the number, just count the opening parenthesis characters, going from left to right.

```
>>> p = re.compile(’(a(b)c)d’)
>>> m = p.match(’abcd’)
>>> m.group(0)
’abcd’
>>> m.group(1)
’abc’
>>> m.group(2)
’b’
```

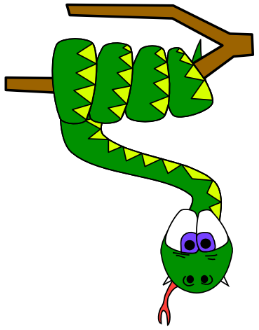


python™

Packages

Packages

- Collection of modules in directory
- Must have __init__.py file
- May contain subpackages
- Import syntax:
  - from P.Q.M import foo; print foo()
  - from P.Q import M; print M.foo()
  - import P.Q.M; print P.Q.M.foo()
  - import P.Q.M as M; print M.foo()   # new



python™

Graphics

DrawingPanel

- To create a window, create a `drawingpanel` and its graphical pen, which we'll call `g` :

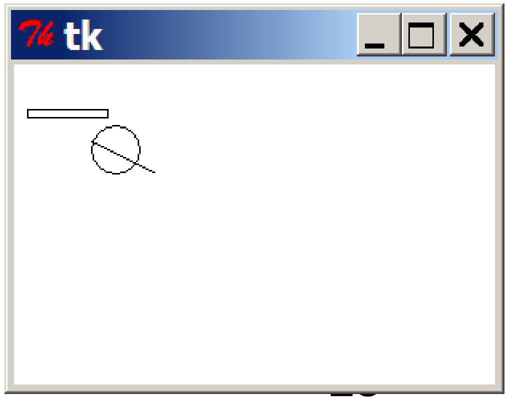
  ```
  from drawingpanel import *
  panel = drawingpanel(width, height)
  g = panel.get_graphics()
  ```

  ... (draw shapes here) ...

  ```
  panel.mainloop()
  ```

- The window has nothing on it, but we can draw shapes and lines on it by sending commands to `g` .
- Example:

  ```
  g.create_rectangle(10, 30, 60, 35)
  g.create_oval(80, 40, 50, 70)
  g.create_line(50, 50, 90, 70)
  ```



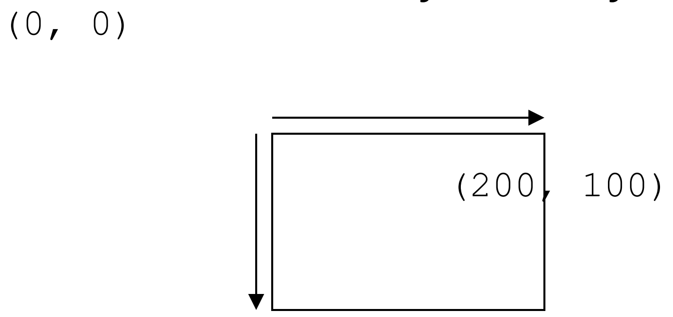
Graphical commands

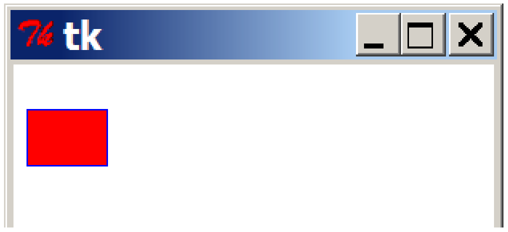
Command	**Description**
`g.create_line(`***x1***`,` ***y1***`,` ***x2***`,` ***y2***`)`	a line between (***x1***, ***y1***), (***x2***, ***y2***)
`g.create_oval(`***x1***`,` ***y1***`,` ***x2***`,` ***y2***`)`	the largest oval that fits in a box with top-left corner at (***x1***, ***y1***) and bottom-left corner at (***x2***, ***y2***)
`g.create_rectangle(`***x1***`,` ***y1***`,` ***x2***`,` ***y2***`)`	the rectangle with top-left corner at (***x1***, ***y1***), bottom-left at (***x2***, ***y2***)
`g.create_text(`***x***`,` ***y***`, text=`*"**text**"*`)`	the given ***text*** at (***x***, ***y***)

- The above commands can accept optional outline and fill colors.

```
g.create_rectangle(10, 40, 22, 65, fill="red", outline="blue")
```

- The coordinate system is y-inverted:



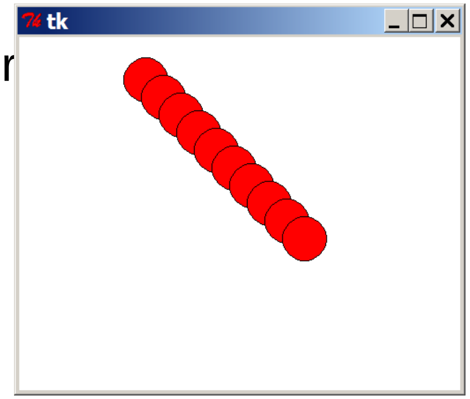


Using the Python Debugger (pdb)

- **Interrupting program execution:**
- You can interrupt the program execution and start the debugger as below
  - import pdb
  - pdb.set_trace()
- The debugger shows a shell that works like the normal Python command line, but with some extra commands:
  - n (next) – execute next statement.
  - s (step) – execute next statement, and descend into functions.
  - l (list) – show source code.
  - c (continue) – continue execution until the next breakpoint.
  - help – print help message.
  - q (quit) – abort the program.
- **Breakpoints:**
  - The command 'b <line number>' sets a breakpoint at the given line.
  - The command 'b' displays all breakpoints set.

Drawing with loops

- We can draw many repetitions of the same item at different x/y positions with `for` loops.
- The x or y assignment expression contains the loop counter pass of the loop, when `i` changes, so does x or y.



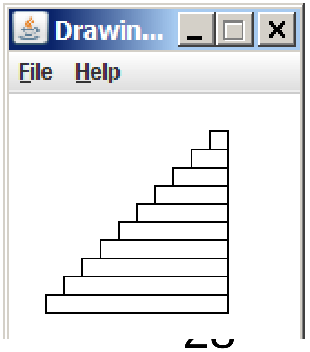
```
from drawingpanel import *

window = drawingpanel(500, 400)
g = window.get_graphics()

for i in range(1, 11):
    x = 100 + 20 * i
    y = 5 + 20 * i
    g.create_oval(x, y, x + 50, y + 50, fill="red")

window.mainloop()
```

- **Exercise:** Draw the figure at right.



Summary

- The following topics are covered so far
  - Regular Expressions
  - MySQL Database connection
  - Simple Django application

Thank you