The current topic: Python

- Introduction
- Object-oriented programming: Python
  - Features, variables, numbers, strings, Booleans, while loops
  - If statements, sequences, functions, modules
  - Dictionaries, command-line arguments, files, classes, inheritance, polymorphism
  - Exceptions, operator overloading, privacy
  - Multiple inheritance, parameters and arguments, list comprehensions
- Next up: Regular expressions, doc strings

- Types and values
- Syntax and semantics
- Functional programming: Scheme
- Exceptions
- Logic programming: Prolog

Announcements

- Lab 1 is due Monday at 10:30 am.
  - Don’t wait till the last minute to figure out how to submit.
  - After submitting, use the command
    
    `submitch326f -l 1`

    (the character after the dash is a lower case L) to see a list of the files you submitted, the size of each file, and the time each file was submitted. Your list of files should include `ex1.py`, `ex2.py`, `ex3.py`, `ex4.py`, `ex5.py`, `ex6.py`, and `MyList.py`.

- Reminder: Term test 1 is on October 6th in GB405, **not in the regular lecture room**.
  - You’re allowed to have one double-sided aid sheet for the test. You must use standard letter-sized (that is, 8.5” x 11”) paper. The aid sheet can be produced however you like (typed or handwritten).
  - Bring your TCard.

Regular expressions

- Formally, a regular expression denotes a language, and the set of languages denoted by regular expressions happens to be equivalent to the set of languages accepted by finite state automata.
  - Don’t worry if this definition seems confusing.

- Informally (and good enough for our purposes): A regular expression (RE) is a pattern that can succeed or fail in matching a string.

- Simple examples (for now we assume that a match has to come at the beginning of a string):

<table>
<thead>
<tr>
<th>RE</th>
<th>string</th>
<th>succeeds?</th>
<th>match</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>aab</td>
<td>Yes</td>
<td>aa</td>
</tr>
<tr>
<td>a</td>
<td>bab</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>ba</td>
<td>bab</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>
### Regular expressions

- A plus means "match one or more of the preceding pattern".

<table>
<thead>
<tr>
<th>RE</th>
<th>string</th>
<th>succeeds?</th>
<th>match</th>
</tr>
</thead>
<tbody>
<tr>
<td>a+</td>
<td>aab</td>
<td>Yes</td>
<td>aa</td>
</tr>
<tr>
<td>a+</td>
<td>bab</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>a+c</td>
<td>abc</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>a+bc</td>
<td>abc</td>
<td>Yes</td>
<td>abc</td>
</tr>
<tr>
<td>a+bc</td>
<td>aaabc</td>
<td>Yes</td>
<td>aaabc</td>
</tr>
<tr>
<td>(ab)+c</td>
<td>ababc</td>
<td>Yes</td>
<td>ababc</td>
</tr>
<tr>
<td>(ab)+cd</td>
<td>ababcd</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>a+b+d</td>
<td>aaabbd</td>
<td>Yes</td>
<td>aaabbd</td>
</tr>
<tr>
<td>a+b+d</td>
<td>bbd</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>a+(ba)+c</td>
<td>ababac</td>
<td>Yes</td>
<td>ababac</td>
</tr>
</tbody>
</table>

- A question mark means "match zero or one of the preceding pattern".

<table>
<thead>
<tr>
<th>RE</th>
<th>string</th>
<th>succeeds?</th>
<th>match</th>
</tr>
</thead>
<tbody>
<tr>
<td>a?</td>
<td>aab</td>
<td>Yes</td>
<td>a</td>
</tr>
<tr>
<td>a?</td>
<td>bab</td>
<td>Yes</td>
<td>(empty string)</td>
</tr>
<tr>
<td>a?c</td>
<td>abc</td>
<td>Yes</td>
<td>abc</td>
</tr>
<tr>
<td>a?bc</td>
<td>abc</td>
<td>Yes</td>
<td>abc</td>
</tr>
<tr>
<td>a?bc</td>
<td>aaabc</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>(ab)?c</td>
<td>cdef</td>
<td>Yes</td>
<td>c</td>
</tr>
<tr>
<td>(ab)?cd</td>
<td>ababcd</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>a?b?d?</td>
<td>dddd</td>
<td>Yes</td>
<td>d</td>
</tr>
<tr>
<td>a?b?d</td>
<td>bbd</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>a?(ba)?c</td>
<td>acbaba</td>
<td>Yes</td>
<td>ac</td>
</tr>
</tbody>
</table>

- A period means "match any single character".

<table>
<thead>
<tr>
<th>RE</th>
<th>string</th>
<th>succeeds?</th>
<th>match</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>aab</td>
<td>Yes</td>
<td>aab</td>
</tr>
<tr>
<td>b.b</td>
<td>bab</td>
<td>Yes</td>
<td>bab</td>
</tr>
<tr>
<td>ac.</td>
<td>abc</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>a.b*</td>
<td>abc</td>
<td>Yes</td>
<td>ab</td>
</tr>
<tr>
<td>a?.c</td>
<td>aaabc</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>c?.e*f</td>
<td>cdef</td>
<td>Yes</td>
<td>cdef</td>
</tr>
<tr>
<td>(ab)?.b</td>
<td>ababcd</td>
<td>Yes</td>
<td>abab</td>
</tr>
<tr>
<td>d..d*e</td>
<td>dddd</td>
<td>Yes</td>
<td>dddd</td>
</tr>
</tbody>
</table>

- To specify an exact number of repetitions, use curly brackets.
  - `{n}` means "match n of the preceding pattern".
  - `{m, n}` means "match between m and n of the preceding pattern".

#### Examples:

<table>
<thead>
<tr>
<th>RE</th>
<th>string</th>
<th>succeeds?</th>
<th>match</th>
</tr>
</thead>
<tbody>
<tr>
<td>.{1,3}</td>
<td>aab</td>
<td>Yes</td>
<td>aab</td>
</tr>
<tr>
<td>b{2}</td>
<td>bab</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>b{2,4}</td>
<td>bbinc</td>
<td>Yes</td>
<td>bb</td>
</tr>
<tr>
<td>a{0,3}</td>
<td>aaabc</td>
<td>Yes</td>
<td>aaa</td>
</tr>
<tr>
<td>(ab){2}.</td>
<td>ababcd</td>
<td>Yes</td>
<td>abab</td>
</tr>
</tbody>
</table>
Regular expressions

• Matching the beginning or end of a string (if we no longer assume that matches have to come at the beginning):
  – The symbol ^ matches the beginning of a string.
  – The symbol $ matches the end of a string.

• Examples (where we don’t assume matches have to come at the beginning):

<table>
<thead>
<tr>
<th>RE</th>
<th>string</th>
<th>succeeds?</th>
<th>match</th>
</tr>
</thead>
<tbody>
<tr>
<td>^...$</td>
<td>aab</td>
<td>Yes</td>
<td>aab</td>
</tr>
<tr>
<td>b.b</td>
<td>ababdb</td>
<td>Yes</td>
<td>bab</td>
</tr>
<tr>
<td>b.b$</td>
<td>ababdb</td>
<td>Yes</td>
<td>bdb</td>
</tr>
<tr>
<td>^b.b</td>
<td>abab</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>^cdf$</td>
<td>abcd</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>

Regular expressions

• Square brackets mean "match one character in this group".
  – [abc] matches any one of the letters a, b, c.
  – [a-z] matches any lower-case letter.
  – [\s\t\n\r\f\v] matches a space, tab, or newline.

• If the first character inside square brackets is the symbol ^, it means "match one character that is not in this group".
  – [^0-9A-Za-z] matches any character that is neither a letter nor a number.

Regular expressions in Python

• To use regular expressions in Python, import the `re` module.

• The functions `re.match()`, `re.search()`, and `re.findall()` look for a match for a given regular expression in a given string.
  – `re.search()` finds the leftmost match
  – `re.match()` only looks for a match at the start of the string
  – `re.findall()` returns a list of all (non-overlapping) matches in the string

• `re.match()` and `re.search()` return `match objects for successful searches, and None for unsuccessful searches.`
Regular expressions in Python

• Match objects provide details about a successful match.

• If \( m \) is a match object produced as a match for a regular expression \( r \) and a string \( s \):
  - \( m\.group() \) is the actual match – that is, the part of string \( s \) that matches expression \( r \).
  - \( m\.start() \) and \( m\.end() \) give the location of the match – that is, the match is \( s[m\.start():m\.end()] \).

Examples:

```python
import re
m1 = re.search('a*b', 'aaaabb')
m1 == None      # False
m1\.group()      # 'aaaab'

m2 = re.search('a*b', 'cab')
m2 == None      # False
m2\.group()      # 'ab'

m3 = re.match('a*b', 'cab')
m3 == None      # True

m4 = re.search('^a*b', 'cab')
m4 == None      # True
```

re.findall() returns a list of all non-overlapping matches.

- This list just stores matches as strings, not as match objects.

```python
m = re.findall('ad*a', 'aaaddaadaaddda')
m   # ['aa', 'adda', 'ada', 'addda']
```

To get match objects for each match, use `re\.finditer()` instead.

- This gives an iteration of match objects.

```python
for i in re\.finditer('ad*a', 'aaaddaadaaddda'):
    print i\.group(),

#Output is: aa adda ada addda
```

Patterns use various special characters and character sequences.

- We've seen examples like `\d` and `\w` where the escape character (the backslash) is used to get a special meaning.
- In other cases, the escape character is used to remove a special meaning from a character.
  - For example, since the symbol `*` has a special meaning, we use `\*` to mean that we actually want to match a star.

And Python itself also uses escapes in strings.

- For example,
  ```python
  print '\\'
  ```
  actually prints a single backslash, not two.
Regular expressions in Python

• Suppose we want to match a backslash in a regular expression.
  – To do this, we need to write `\` in the regular expression.
  – Then, when writing the regular expression as a Python string, we actually need to write `\\`, since within a Python string each backslash must be escaped.
  • This becomes confusing, especially for more complex regular expressions.

• To avoid confusion, use raw strings to write regular expressions. These are string literals preceded by the letter `r`. Within a raw string, backslashes have no special meaning (unless they immediately precede a quotation mark).
  – For example,
    ```python
    print r'\n'
    ```
    actually prints `\n`, not a single slash followed by a newline.

Regular expressions in Python

• The first step performed by `re.search()`, `re.match()`, and `re.findall()` is to translate the regular expression it’s given into an internal data structure.

• If your regular expression is going to be used more than once, it makes sense to compile it.
  – This produces a regular expression object, which you can use any number of times to perform matches by calling its `search()`, `match()`, and `findall()` methods.

```python
r = re.compile(r'a*b')
m1 = r.search('aab')
m1.group()   # 'aab'
m2 = r.match('bdca')
m2.group()   # 'b'
```

Regular expressions in Python

• Regular expressions are greedy, in the sense that each component of a regular expression will "eat up" as much of the search string as it can.

```python
remoney = re.compile(r'(.*)\d+\d\.)')
r = remoney.search('Current balance = 12345.67')
r.groups()   # ('Current balance = ', '12345.67')
```

• Observe that when a regular expression includes parts that are enclosed in brackets (these parts are called subpatterns), we can use the match object’s `groups()` method to find the parts of the match corresponding to each subpattern.

• In the example above, the first subpattern `('.*)` matches as much as possible while allowing the success of the second pattern. So it has "eaten up" all digits before the decimal point, even though that’s not really what we wanted.

```python
remoney = re.compile(r'(.*)\d+\d\.)')
r = remoney.search('Current balance = 12345.67')
r.groups()   # ('Current balance = ', '12345.67')
```
Regular expressions in Python

- We can use the `re.sub()` and `re.subn()` functions (and the `sub()` and `subn()` methods of regular expression objects) to replace matches with a different string.

  ```python
  s = re.sub('a.a', 'hi!', 'acabcdaaaffada')
  s  # hi!bcdhi!ffhi!
  ```

- We can also specify the maximum number of replacements to perform (where replacements are performed from left to right).

  ```python
  s = re.sub('a.a', 'hi!', 'acabcdaaaffada', 2)
  s  # hi!bcdhi!ffada
  ```

- To find out how many replacements were performed, use `re.subn()`. This returns a tuple consisting of the new string and the number of replacements made.

  ```python
  s, n = re.subn('a.a', 'hi!', 'acabcdaaaffada')
  s  # hi!bcdhi!ffhi!
  n  # 3
  ```

Doc strings

- **Doc strings** are the standard way to provide documentation in Python.
  - The `help()` function (that we saw earlier) displays the contents of doc strings.

- Doc strings may appear as the first line of code inside:
  - Modules
  - Classes
  - Functions
  - Methods

- Doc strings are written as string literals.
  - To use multi-line doc strings, we need to use the multi-line version of string literals. These begin and end with """" (three double-quotes).
  - For consistency, it’s a good idea to always begin and end doc strings with """", even if they only use a single line. This also makes it easier to add lines to the doc string later on.

- A simple example:

  ```python
  def f(x, y):
      """Returns the sum of x and y."""
      return x+y
  ```

  - Note that the first line of the doc string has to have the same indentation as the rest of the function body.

  - To view the documentation:

    ```python
    help(f)
    ```

    Output:
    Help on function f in module __main__:
    f(x, y)
    Returns the sum of x and y.
Doc strings

- Doc strings are stored as an attribute called \_\_doc\_.

  \ f.__doc__  # 'Returns the sum of x and y.'

- A multi-line example:

  \ def g(x, y):
  """Returns x divided by y.
  x -- Any integer.
  y -- Any non-zero integer.
  """

  \ return x/y

Doc strings

- Documenting a simple class using doc strings:

  \ class A(object):
  """Represents something..."

  \ def \_\_init\_(self):
  """Constructor.""
  \ pass

  \ def m(self, x):
  """Does something.
  x -- Some parameter.
  """
  \ pass

Exercises

- Add doc strings to the \texttt{NewFibonacci} class from previous exercises. Then, call \texttt{help(NewFibonacci)} to see your doc strings in action.

- Write a function that replaces all IP addresses in a given file with 0.0.0.0. (This is roughly the sort of thing a website like Google might do to anonymize its logs in order to protect its users' privacy.) Use regular expressions to find IP addresses. For the purposes of this exercise, an IP address is a string of the form \texttt{<number>.<number>.<number>.<number>}, where each \texttt{<number>} has between 1 and 3 digits. The function should take the names of the original file and the new file as parameters.