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

    ```
    submitcsc326f -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):

RE	**string**	**succeeds?**
a	aab	Yes
a	bab	No
ba	bab	Yes

Regular expressions

- A star means "match any number (including 0) of the preceding pattern".

RE	**string**	**succeeds?**	**match**
a*	aab	Yes	aa
a*	bab	Yes	(empty string)
a*	bbb	Yes	(empty string)
a*c	abc	No	
a*c	cde	Yes	c
(ab)*c	ababc	Yes	ababc
(ab)*cd	ababdcd	No	
a*b*d	aaabbbd	Yes	aaabbbd
a*b*d	bbd	Yes	bbd
a*ba*b*	bac	Yes	ba

Regular expressions

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

RE	string	succeeds?	match
a+	aab	Yes	aa
a+	bab	No	
a+c	abc	No	
a+bc	abc	Yes	abc
a+bc	aaabc	Yes	aaabc
(ab)+c	ababc	Yes	ababc
(ab)+cd	ababdcd	No	
a+b+d	aaabbbd	Yes	aaabbbd
a+b+d	bbd	No	
a+(ba)+c	ababac	Yes	ababac

Regular expressions

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

RE	string	succeeds?	match
a?	aab	Yes	a
a?	bab	Yes	(empty string)
a?c	abc	No	
a?bc	abc	Yes	abc
a?bc	aaabc	No	
(ab)?c	cdef	Yes	c
(ab)?cd	ababdcd	No	
a?b?d?	dddd	Yes	d
a?b?d	bbd	No	
a?(ba)?c	acbaba	Yes	ac

Regular expressions

- A period means "match any single character".

RE	string	succeeds?	match
...	aab	Yes	aab
b.b	bab	Yes	bab
ac.	abc	No	
a.b*	abc	Yes	ab
a?..c	aaabc	No	
c?.e*f	cdef	Yes	cdef
(ab)?.b	ababdcd	Yes	abab
d..de*	dddd	Yes	dddd

Regular expressions

- 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:

RE	string	succeeds?	match
.{1,3}	aab	Yes	aab
b{2}	bab	No	
b{2,4}	bbbac	Yes	bbb
a{0,3}	aaabc	Yes	aaa
(ab){2}.	ababdcd	Yes	ababd

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):

RE	string	succeeds?	match
`^...$`	aab	Yes	aab
`b.b`	ababdb	Yes	bab
`b.b$`	ababdb	Yes	bdb
`^b.b`	abab	No	
`^cdf$`	abcdf	No	

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.
  - `[A-Za-z]` matches any letter.
  - `[ \t\n]` 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

- Special characters:
  - \s matches any whitespace character
    - equivalent to `[ \t\n\r\f\v]`
    - \t is a tab, \r is a carriage return, \f is a form feed, \v is a vertical tab
  - \d matches any digit
    - equivalent to `[0-9]`
  - \w matches any letter, number, or underscore
    - equivalent to `[0-9A-Za-z_]`
  - \\ matches a single backslash

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()]`.

Regular expressions in Python

- Examples:

```
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
```


Regular expressions in Python

- `re.findall()` returns a list of all non-overlapping matches.
  - This list just stores matches as strings, not as match objects.

```
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.

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


#Output is: aa adda ada addda
```


Regular expressions in Python

- 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,
    `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,
    `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 that 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.

```
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.

```
remoney = re.compile(r'(.*)(\d*\.\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 (there 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.

Regular expressions in Python

- To solve this problem, we can use the non-greedy version of the `*` operation.
  - The non-greedy version is `*?`.
  - Similarly, the non-greedy version of + is `+?`, and the non-greedy version of `?` is `??`.
  - The non-greedy versions try to match as little as possible while allowing for the success of the pattern.

```
remoney = re.compile(r'(.*?)(\d*\.\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.

```
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).

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


Regular expressions in Python

- 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.

```
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.

Doc strings

- A simple example:

```
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:

```
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
```


Doc strings

- Then, the output of `help(A)` is:

```
Help on class A in module __main__:

class A(__builtin__.object)
 |  Represents something...
 |
 |  Methods defined here:
 |
 |  __init__(self)
 |      Constructor.
 |
 |  m(self, x)
 |      Does something.
 |
 |      x -- Some parameter.
 |
 |
----------------------------------------------------------------
```


Exercises

- Add doc strings to the `NewFibonacci` class from previous exercises. Then, call `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 `<number>.<number>.<number>.<number>`, where each `<number>` has between 1 and 3 digits. The function should take the names of the original file and the new file as parameters.