LECTURE 8

The Standard Library Part 2: re, copy, and itertools

THE STANDARD LIBRARY: RE

The Python standard library contains extensive support for *regular expressions*.

Regular expressions, often abbreviated as regex, are special character sequences that represent a set of strings. They are a concise way to express a set of strings using formal syntax.

We’ll start by learning Python’s syntax for regular expressions and then we’ll see how we can use the re module to compile and use regular expression objects.

THE STANDARD LIBRARY: RE

The simplest regular expression contains a single ordinary character.

```
'A'      'b'      '0'      '@'
```

This character simply “matches” itself – that is, these regular expressions define the set containing only the character itself.

We can also concatenate ordinary characters to create a string longer than one character.

```
'word'      '@abc123'
```

The regular expression ‘word’ simply defines the set that contains the string ‘word’ and the regular expression ‘@abc123’ simply defines the set that contains the string ‘@abc123’.

THE STANDARD LIBRARY: RE

The only characters that don't match themselves are the special characters, or metacharacters.

```
. ^ $ * + ? { } [ ] \ | ( )
```

- The dot (`.`) metacharacter matches any ordinary character except the newline character.

```
'.' = {"a", "b", "c", ..., "A", "B", ... "@", ...}
```

- The caret (`^`) metcharacter matches any string that starts with the following sequence of characters.

```
'^a' = {"a", "apple", "air", "age", "armor", "a new day",...}

'^up' = {"up", "up and away", "upper", "upon a hill", ...}
```

THE STANDARD LIBRARY: RE

- The $ metacharacter matches any string that ends with the preceding sequence of characters.

```
'ear$' = {"ear", "clear", "top gear", ...}
```

- The * metacharacter matches 0 or more instances of the preceding regular expression.

```
'b*' = {"", "b", "bb", "bbb", ...}
'ab*' = {"a", "ab", "abb", "abbb", ...}
```

Note the behavior here – the preceding RE is ‘b’ because it is the simplest possible preceding RE. To force a grouping, we can use parentheses.

```
'(ab)*' = {"", "ab", "abab", "ababab", ...}
```

THE STANDARD LIBRARY: RE

- The + metacharacter works just the like * metacharacter except it matches one or more instances of the preceding regular expression.

```
'(ab)+' = {"ab", "abab", "ababab", ...}
```

- The ? metacharacter matches either zero or one instances of the preceding regular expression.

```
'(ab)?' = {"", "ab"}
```

THE STANDARD LIBRARY: RE

- The `{m}` metacharacters specify that exactly *m* copies of the previous regular expression should be matched.

```
'b{4}' = {"bbbb"}
'ab{4}' = {"abbbb"}
'(ab){4}' = {"abababab"}
```

- We can also use `{m,n}` specify a range from *m* to *n* copies of the previous regular expression. We can also leave either argument out to specify an unbounded end of the range.

```
'b{,4}' = {"", "b", "bb", "bbb", "bbbb"}
'ab{4,}' = {"abbbb", "abbbbb", "abbbbbb", ...}
```

THE STANDARD LIBRARY: RE

- The `[ ]` metacharacters are used to indicate a set of characters.
- List characters individually: `'[abc]' = {"a", "b", "c"}`
- Create a range: `'[a-zA-Z]' = {"a", "A", "b", "B", ..., "z", "Z"}`
  `'[0-9]' = {"0", "1", "2", ..., "9"}`
- Complement the set: `'[^aeiou]' = {"b", "c", "d", "f", "g", ...}`
- Note: special characters lose their meaning or take on new meaning inside of sets.
- You can use `\` to escape the special characters in a set. `'[a\-z]' = {"a", "-", "z"}`

THE STANDARD LIBRARY: RE

- The `|` metacharacter is used to implement alternation. It represents either the regular expression on the left side OR the regular expression on the right side.

```
'a|b' = {"a", "b"}
'(hello)|(goodbye)' = {"hello", "goodbye"}
```

There are a lot of details to the regular expression mini-language that we haven't covered – some we'll cover later, some are left up to you to look up. ☺

Note that, theoretically speaking, you can express *any* set of strings using only `*`, `()`, and `|` in your regular expression. So, we definitely have enough to match some strings.

THE STANDARD LIBRARY: RE

To use a regular expression, we first have to compile it. This creates a pattern object which has methods for searching and matching.

The `re.compile(pattern, flags=0)` method compiles a regular expression pattern into a pattern object. There are a number of optional flags that can be used to affect how the pattern object is created.

```
>>> import re
>>> p = re.compile('[1-9][0-9]*')
>>> p
<_sre.SRE_Pattern object at 0x...>
```

THE STANDARD LIBRARY: RE

Some of the available compile flags:

- `re.I` – ignore case.
- `re.S` – the dot metacharacter also matches newline.
- `re.M` – The caret metacharacter and the '$' metacharacter match not only the beginning and end of strings, respectively, but the beginning and end of each newline-delimited portion of the string.

Use a | to separate the selected compile flags. What kinds of strings does the following compiled pattern object match?

```
    re.compile('^A', re.I | re.M)
```

THE STANDARD LIBRARY: RE

One little quirk to note about regular expressions comes from the fact that we express regular expression patterns as strings.

One of the built-in special characters for a regular expression is \ which allows for escaping or special forms. However, note that \ also has a special meaning in Python strings.

Say we want to create a regular expression to match the string “`.\temp.txt`”. We must first escape the special characters. This gives us “`\.\\temp\.txt`”. A Python string will interpret this as “`.\temp.txt`” because \ is the escape character. To prevent Python from removing the backslashes, we must escape all of them! Finally, we have “`\\.\\\\temp\\.txt`” as our regular expression string.

THE STANDARD LIBRARY: RE

To avoid this excessive parentheses issue, we can express our regular expression as a raw string by simply appending ‘r’ to the front of it.

```
      r“\.\\temp\.txt”
```

Much better! Because this is a raw string, Python will not try to interpret the special characters before passing the string into the compile method.

THE STANDARD LIBRARY: RE

So, using the `re.compile(pattern, flags=0)` method, we can create a pattern object out of the raw string representation of our regular expression.

The first pattern object method we'll cover is the `match(string)` method. The `match(string)` method determines if the regular expression matches from the beginning of the *string*.

THE STANDARD LIBRARY: RE

The `match(`*string*`)` method determines if the regular expression matches from the beginning of the *string*. Note the creation of a Match object when there is a match. What does `p.match()` return when the string doesn't match?

```
>>> import re
>>> re.compile(r'A', re.I)
<_sre.SRE_Pattern object at 0x7f763fe5fb30>
>>> p = re.compile(r'A', re.I)
>>> p.match("A mind is a terrible thing to waste.")
<_sre.SRE_Match object at 0x7f763fedc440>
>>> p.match("abcd")
<_sre.SRE_Match object at 0x7f763fedc3d8>
>>> p.match("Hello, Alice!")
>>>
```

THE STANDARD LIBRARY: RE

Note that the following two code snippets are equivalent. Compiling your regular expressions into Pattern objects is preferable, especially when the regular expression is used multiple times throughout the execution of the program.

```
>>> import re
>>> p = re.compile(r'A', re.I)
>>> p.match("A mind is a terrible thing to waste.")
```

```
>>> import re
>>> re.match(r'A', "A mind is a terrible thing to waste.", re.I)
```

THE STANDARD LIBRARY: RE

The search(*string*) method scans the *string*, looking for any instance where the regular expression can be matched.

```
>>> p = re.compile(r'[1-9][0-9]*')
>>> p.search("My office number is 205A.")
<_sre.SRE_Match object at 0x7f763fedc3d8>
>>> p.search("11:00 to 12:15 is our class time.")
<_sre.SRE_Match object at 0x7f763fedc648>
>>> p.search("Python regular expressions are neat!")
>>>
```

THE STANDARD LIBRARY: RE

As we just saw, the `match()` and `search()` methods return a Match object. Match objects have many methods for accessing information about the matched string.

Match objects always evaluate to `True`, so you can use the return value of the `match` and `search` methods to perform a Boolean test of whether there was a match or not.

The first important match method we need to know is group([group1, …]). This function returns the subgroups of the match. The ( ) metacharacters are used to create subgroups. The entire match is always *0*, which the parenthesized subgroups are given the identifiers *1*, *2*, *3*, etc in order.

THE STANDARD LIBRARY: RE

```
>>> p = re.compile(r'\$([1-9][0-9]*)\.([0-9]{2})')
>>> m = p.search("This book costs $10.95.")
>>> m.group()
'$10.95'
>>> m.group(0)
'$10.95'
>>> m.group(1)
'10'
>>> m.group(2)
'95'
>>> m.group(1,2)
('10', '95')
>>> m.groups() # Returns all of the groups as a tuple
('10', '95')
```

THE STANDARD LIBRARY: RE

You can also identify subgroups by keyword rather than index. This is done using the `(?P<group_name>RE)` syntax. Consider the example below.

```
>>> p = re.compile(r'\$(?P<dollars>[1-9][0-9]*)\.(?P<cents>[0-9]{2})')
>>> m = p.search("This book costs $10.95.")
>>> m.group()
'$10.95'
>>> m.group('dollars')
'10'
>>> m.group('cents')
'95'
```

THE STANDARD LIBRARY: RE

Note: The `*`, `+`, `?` and `{}` metacharacters are all *greedy* – that is, they will try to match as many times as possible. Use a following `?` to force them to match minimally.

```
>>> p = re.compile(r'<.*>')
>>> m = p.search("<span>Here's some content.</span>")
>>> m.group()
"<span>Here's some content.</span>"
>>> p = re.compile(r'<.*?>')
>>> m = p.search("<span>Here's some content.</span>")
>>> m.group()
'<span>'
```

THE STANDARD LIBRARY: RE

That's it for the basics of regular expressions in Python. There are a lot of additional methods, customizations, and quirks that you might want to know about if you intend on using regular expressions for a complex application. Check here for more info.

For now, let's check out an example application using regular expressions. We'll be attempting the Baby Names exercise. I encourage you all to read the write-up and suggested steps for solving the problem. Being able to develop incrementally is an incredibly important skill. It is not wise to attempt to write a complete application, however small it may be, and test only after you've written all the code.

THE STANDARD LIBRARY: COPY

The next standard library module we'll look at is a relatively small module called copy. The copy modules provides the methods and exceptions necessary to create copies of Python objects.

Before we continue, there is one point that needs to be crystal clear. Python performs *assignment* by creating a binding between a *name* (e.g. `x`, `mylist`, `sum_of_squares`) and an *object* (e.g. an int object, a list object, a function object).

```
x = 2
y = x
```

In this example, we are not creating a copy of x, we are creating another name binding to the same int object that holds 2.

THE STANDARD LIBRARY: COPY

Some objects have built-in methods for creating copies. For example,

```
y = x[:] # y is a copy of x
```

Universally, however, we can use the `copy.copy(x)` method. This will return a *shallow* copy of the object bound to the name *x*. A shallow copy will only create a new compound object that references the original nested objects.

THE STANDARD LIBRARY: COPY

Notice that making a change to the compound structure, as we do with

```
    y[0] = 1
```

does not affect the original.

However, if we modify the nested elements, the change will be reflected in the original list.

```
>>> import copy
>>> x = [[1, 2], [3, 4]]
>>> y = copy.copy(x)
>>> z = copy.copy(x)
>>> y[0] = 1
>>> y
[1, [3, 4]]
>>> x
[[1, 2], [3, 4]]
>>> z[0][1] = 5
>>> z
[[1, 5], [3, 4]]
>>> x
[[1, 5], [3, 4]]
```

THE STANDARD LIBRARY: COPY

To create a true copy, we can use the `copy.deepcopy(x)` method, which recursively copies the objects of x.

Notice that even when modifying the nested objects, we do not affect the original. This is because even the nested objects are copies of the original nested objects.

```
>>> import copy
>>> x = [[1, 2], [3, 4]]
>>> y = copy.deepcopy(x)
>>> z = copy.deepcopy(x)
>>> y[0] = 1
>>> y
[1, [3, 4]]
>>> x
[[1, 2], [3, 4]]
>>> z[0][1] = 5
>>> z
[[1, 5], [3, 4]]
>>> x
[[1, 2], [3, 4]]
```

THE STANDARD LIBRARY: COPY

Behind the scenes,

```
y = copy.copy(x)
z = copy.deepcopy(x)
```

makes use of the following calls:

```
y = x.__copy__()
z = y.__deepcopy__()
```

So, to be able to use the copy module with your custom class, just implement the `__copy__()` and `__deepcopy__()` methods to return the appropriate object.

THE STANDARD LIBRARY: ITERTOOLS

The itertools module is inspired by functional programming languages such as Haskell and SML. The methods provided are fast and memory-efficient and, together, form an “iterator algebra” for constructing specialized iterators.

We’ll start with the infinite iterators – these clearly are created by generators since it would be impossible to store an “infinite” dataset in memory!

THE STANDARD LIBRARY: ITERTOOLS

The Infinite Iterators:

- `itertools.count(`*`start=0, step=1`*`)` – creates an iterator that returns evenly-spaced values starting with *start*.
- `itertools.cycle(`*`iterable`*`)` – creates an iterator returning elements from the *iterable* and saving a copy of each. When the iterable is exhausted, return elements from the saved copy, repeating indefinitely.
- `itertools.repeat(`*`object`*`[,` *`times`*`])` -- creates an iterator that returns *object* over and over again. Runs indefinitely unless the times argument is specified. Commonly used with iterators that require multiple copies of invariant values.

THE STANDARD LIBRARY: ITERTOOLS

```
>>> import itertools
>>> for i in itertools.count(10, 2):
...     print i
...     if i > 19:
...         break
...
10
12
14
16
18
20
```

THE STANDARD LIBRARY: ITERTOOLS

```
>>> import itertools
>>> counter = 0
>>> for i in itertools.cycle([1, 2, 3]):
...     print i,
...     counter = counter + 1
...     if counter > 12:
...         break
...
1 2 3 1 2 3 1 2 3 1 2 3 1
```

THE STANDARD LIBRARY: ITERTOOLS

```
>>> import itertools
>>> counter = 0
>>> for i in itertools.repeat("hi", 5):
...     print i,
...
hi hi hi hi hi
```

THE STANDARD LIBRARY: ITERTOOLS

The following itertools iterators terminate on the shortest sequence:

- `itertools.chain(*iterables)` – creates an iterator that returns elements from the first iterable until it is exhausted, then proceeds to the next iterable, until all of the iterables are exhausted.
- `itertools.izip(*iterables)` – creates an iterator that aggregates elements from each of the iterables.
- `itertools.imap(function, *iterables)` – creates an iterator that computes the function using arguments from each of the iterables. If function is set to None, then imap() returns the arguments as a tuple.

THE STANDARD LIBRARY: ITERTOOLS

```
>>> from itertools import *
>>> for i in chain(['a', 'b', 'c'], [1, 2, 3]):
...     print i
...
a
b
c
1
2
3
```

THE STANDARD LIBRARY: ITERTOOLS

```
>>> from itertools import *
>>> for i in izip(['a', 'b', 'c'], [1, 2, 3]):
...     print i
...
('a', 1)
('b', 2)
('c', 3)
```

THE STANDARD LIBRARY: ITERTOOLS

```
>>> from itertools import *
>>> for i in imap(lambda x,y: (x, y, x*y), xrange(5), xrange(5,10)):
...     print '{} * {} = {}'.format(*i)
...
0 * 5 = 0
1 * 6 = 6
2 * 7 = 14
3 * 8 = 24
4 * 9 = 36
```

THE STANDARD LIBRARY: ITERTOOLS

```
>>> import itertools as it
>>> for i in it.imap(pow, xrange(10), it.repeat(2)):
...     print i,
...
0 1 4 9 16 25 36 49 64 81
```

THE STANDARD LIBRARY: ITERTOOLS

Combinatoric generators:

- `itertools.permutations(iterable[, r])` – returns successive *r* length permutations of elements in the *iterable*.
- `itertools.combinations(iterable, r)` – returns *r* length subsequences of elements from the input *iterable*.

THE STANDARD LIBRARY: ITERTOOLS

```
>>> from itertools import *
>>> for i in permutations('ABC', 2):
...     print i,
...
('A', 'B') ('A', 'C') ('B', 'A') ('B', 'C') ('C', 'A') ('C', 'B')
>>> for i in combinations([1,2,3,4], 2):
...     print i,
...
(1, 2) (1, 3) (1, 4) (2, 3) (2, 4) (3, 4)
```

LEXICOGRAPHIC PERMUTATIONS

A permutation is an ordered arrangement of objects. For example, 3124 is one possible permutation of the digits 1, 2, 3 and 4. If all of the permutations are listed numerically or alphabetically, we call it lexicographic order. The lexicographic permutations of 0, 1 and 2 are:

012   021   102   120   201   210

What is the millionth lexicographic permutation of the digits 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9?

LEXICOGRAPHIC PERMUTATIONS

```
from itertools import permutations
import string

def find_perm(num):
    for i, p in enumerate(permutations(string.digits), start=1):
        if i == num:
            return ''.join(p)

if __name__ == "__main__":
    print "The one millionth permutation is", find_perm(1000000)
```

LEXICOGRAPHIC PERMUTATIONS

```
from itertools import permutations
import string

def find_perm(num):
    for i, p in enumerate(permutations(string.digits), start=1):
        if i == num:
            return ''.join(p)

if __name__ == "__main__":
   print "The one millionth permutation is", find_perm(1000000)
```

```
$ python lex.py
The one millionth permutation is 2783915460
```

LEXICOGRAPHIC PERMUTATIONS

```python
from itertools import permutations, islice
import string

def find_perm(num):
    return ''.join(next(islice(permutations(string.digits), num-1, num)))

if __name__ == "__main__":
    print "The one millionth permutation is", find_perm(1000000)
```

```
$ python lex.py
The one millionth permutation is 2783915460
```

LOOP LIKE A NATIVE

You are encouraged to check out Ned Batchelder's talk "Loop Like A Native: while, for, iterators, generators" given at PyCon '13.