Chapter 13 - Regular Expressions

CS 172 - Computer Programming 2
Lanzhou University

These slides use many elements provided in the main bibliographic reference for these lectures:

Programming in Python 3
A Complete Introduction to the Python Language,
2nd Edition,
Mark Summerfield

Outline

1 Characters and Character Classes

2 Quantifiers

3 Grouping

Regular Expressions

- A regular expression is a compact notation for representing a collection of strings
  - Regular expressions are also known as regexes
- One regular expression can represent an unlimited number of strings
  - which makes this such a powerful mechanism
- Regular expressions are used in:
  1. Parsing
  2. Searching
  3. Searching and replacing
  4. Splitting strings
  5. Validation

Regular Expressions

In this chapter

- We take a hands-on approach to exploring regular expressions
  - and their associated constructions and concepts
- We essentially explore the use of the `re` module
  - through which we can seamlessly create and use regular expressions
- We therefore focus in Python
  - But keep in mind that regexes are a general concept
  - Which is many times useful in programming

Outline

1. Characters and Character Classes
2. Quantifiers
3. Grouping

Characters and Character Classes

- The simplest expressions are just literal characters such as a or 5
- Each regex as these match just a single character
- For instance, the regex `CS172` matches one occurrence of `C`, followed by one `S`, followed by one `1`, followed by one `7`, followed by one `2`
- Most characters can be used as literals
- We can do this in Python:

```
>>> match = re.match(’CS172’, ’CS172 is cool’)
>>> match.group(0) if match else print("no match")
’CS172’
>>> match = re.match(’CS172’, ’I love CS172!!!’)
>>> match.group(0) if match else print("no match")
no match
```

- `match` will try to match the given regex (first parameter) in the beginning of the second parameter (a string)
- `search` will try to match the regex anywhere in the string:

```
>>> match = re.search(’CS172’, ’I love CS172!!!’)
>>> match.group(0) if match else print("no match")
’CS172’
```

Characters and Character Classes

- But some are *special characters*
  - Which are symbols in the regex language
  - And so must be escaped by preceding them with a backslash \ to use them as literals
  - Special characters are `\ . ^ $ ? + { } [ ] ( ) |`
- For instance, to match with, e.g., +, since + is an operator of the regex language
  - we must escape it (include a \ before it) in the regex

```
>>> match = re.match(’\+’, ’+a’)
>>> match.group(0) if match else print("no match")
’+’
```

Characters and Character Classes

In summary

- We are trying to match *one occurrence* of a literal
  - in the beginning of a string
- Function `re.match(r, s, f)`
  - returns a match object if the regex `r` matches at the start of string `s`
    - otherwise returns `None`
  - Flag(s) `f` can optionally be passed
- Function `m.group(g, ...)` works on a match object `m`
  - Returns the numbered capture group `g`
    - the whole match is group 0

Characters and Character Classes

- We can also try to match with any character of a set of characters
  - this is achieved using a *character class*, i.e.
  - one or more characters enclosed in square brackets

```
>>> match = re.match(’[aeiou]’, ’abc’)
>>> match.group(0) if match else print("no match")
’a’
>>> match = re.match(’[aeiou]’, ’ebc’)
>>> match.group(0) if match else print("no match")
’e’
>>> match = re.match(’[aeiou]’, ’xbc’)
>>> match.group(0) if match else print("no match")
no match
```

Characters and Character Classes

- It is possible to negate the meaning of a character class
  - By following the opening bracket with a caret
    - so, e.g., [ˆ0123456789] matches any character that is *not* a digit

```
>>> match = re.match(’[ˆ0123456789]’, ’abc’)
>>> match.group(0) if match else print("no match")
’a’
>>> match = re.match(’[ˆ0123456789]’, ’7bc’)
>>> match.group(0) if match else print("no match")
no match
```

- Inside a character class, special characters lose their special meaning
  - Except for \
  - And for ˆ, which acquires the meaning of negation if it is the 1st character in the character class
    - otherwise it is simply a literal caret

Outline

1 Characters and Character Classes

2 Quantifiers

3 Grouping

Quantifiers

- A quantifier has the form `{m, n}`, where `m` and `n`
  - are the minimum and maximum times the expression the quantifier applies to must match

```
>>> match = re.match(’e{1,1}e{1,1}’, ’eel’)
>>> match.group(0) if match else print("no match")
’ee’
>>> match = re.match(’e{1,2}’, ’eel’)
>>> match.group(0) if match else print("no match")
’ee’
>>> match = re.match(’e{3,5}’, ’eel’)
>>> match.group(0) if match else print("no match")
no match
```

- There are also convenient shorthands
  - if only one number is given in the quantifier it is taken to be both the minimum and the maximum

```
>>> match = re.match(’e{2}’, ’eel’) # the regex is the same as e{2,2}
>>> match.group(0) if match else print("no match")
’ee’
```

Quantifiers

- If no quantifier is explicitly given
  - it is assumed to be one, i.e., {1, 1} or {1}

```
>>> match = re.match('ee', 'eel') # the regex is the same as e{1}e{1} or e{2}
>>> match.group(0) if match else print("no match")
'ee'
```

- The {0,1} quantification is so often used
  - that it has its own shorthand form, ?

```
>>> match = re.match('a?eiou', 'aeiou')
>>> match.group(0) if match else print("no match")
'aeiou'
>>> match = re.match('a?eiou', 'eiou')
>>> match.group(0) if match else print("no match")
'eiou'
```

Quantifiers

- There is also +, which stands for `{1,n}`
  - so it means *at least one*

```
>>> match = re.match(’a+eiou’, ’aeiou’)
>>> match.group(0) if match else print("no match")
’aeiou’
>>> match = re.match(’aaaaa+eiou’, ’aeiou’)
>>> match.group(0) if match else print("no match")
no match
>>> match = re.match(’a+eiou’, ’aaaaaeiou’)
>>> match.group(0) if match else print("no match")
’aaaaaeiou’
>>> match = re.match(’a+eiou’, ’eiou’)
>>> match.group(0) if match else print("no match")
no match
```

- Finally, * stands for `{0,n}`
  - so it means *any number of*

```
>>> match = re.match(’a*eiou’, ’eiou’)
>>> match.group(0) if match else print("no match")
’eiou’
>>> match = re.match(’a*eiou’, ’aaaaeiou’)
>>> match.group(0) if match else print("no match")
’aaaaeiou’
```

Outline

1. Characters and Character Classes
2. Quantifiers
3. Grouping

Grouping

- Sometimes we want a quantifier to apply to several expressions
  - We can group expressions using ()

```
>>> match = re.match(’(abc)*def’, ’def’)
>>> match.group(0) if match else print("no match")
’def’
>>> match = re.match(’(abc)*def’, ’abcabcdef’)
>>> match.group(0) if match else print("no match")
’abcabcdef’
>>> match = re.match(’(abc)*def’, ’abdef’)
>>> match.group(0) if match else print("no match")
no match
```

- What would the following code produce? Why?

```
>>> match = re.match(’(a(bc)+d)*ef’, ’abcbcdef’)
>>> match.group(0) if match else print("no match")
```

Grouping

- Sometimes we want a quantifier to apply to several expressions
  - We can group expressions using ()

```
>>> match = re.match(’(abc)*def’, ’def’)
>>> match.group(0) if match else print("no match")
’def’
>>> match = re.match(’(abc)*def’, ’abcabcdef’)
>>> match.group(0) if match else print("no match")
’abcabcdef’
>>> match = re.match(’(abc)*def’, ’abdef’)
>>> match.group(0) if match else print("no match")
no match
```

- What would the following code produce? Why?

```
>>> match = re.match(’(a(bc)+d)*ef’, ’abcbcdef’)
>>> match.group(0) if match else print("no match")
```

- `’abcbcdef’`

Grouping

- We can also use alternatives using |
  - which is useful when we want to match any of several different alternatives

```
>>> match = re.match(’aircraft|airplane|jet’, ’jet’)
>>> match.group(0) if match else print("no match")
’jet’
>>> match = re.match(’aircraft|airplane|jet’, ’aircraft’)
>>> match.group(0) if match else print("no match")
’aircraft’
>>> match = re.match(’aircraft|airplane|jet’, ’boat’)
>>> match.group(0) if match else print("no match")
no match
```

- What would the following code produce? Why?

```
>>> match = re.match(’(ab)+|(cd)*’, ’abcd’)
>>> match.group(0) if match else print("no match")
```

Grouping

- We can also use alternatives using |
  - which is useful when we want to match any of several different alternatives

```
>>> match = re.match(’aircraft|airplane|jet’, ’jet’)
>>> match.group(0) if match else print("no match")
’jet’
>>> match = re.match(’aircraft|airplane|jet’, ’aircraft’)
>>> match.group(0) if match else print("no match")
’aircraft’
>>> match = re.match(’aircraft|airplane|jet’, ’boat’)
>>> match.group(0) if match else print("no match")
no match
```

- What would the following code produce? Why?

```
>>> match = re.match(’(ab)+|(cd)*’, ’abcd’)
>>> match.group(0) if match else print("no match")
```

- `’ab’` because it either recognizes `(ab)+` OR `(cd)*` (but only if the first alternative fails)
  - in this case, the string starts with `ab` and thus the first alternative matches