Python - regular expressions

Janusz Jabłonowski

Searching for a pattern (string) in a text (string)

- Notions
  - a text and a pattern
  - lengths: n for text, m for pattern
- It can be done by iterating over the text
- The naive algorithm has computational cost of O(n*m)
- There are faster algorithms
  - Knuth–Morris–Pratt algorithm (requires preprocessing of the pattern and additional table of size O(m), the cost is O(m+n))
  - Rabin-Karp (uses hashes, on average O(n+m), but worst O((n-m)*m)
  - many others ...
- This task is so fundamental, that there should be something built-in

Searching for a pattern (string) in a text (string)

- We can use Python string operations
- The **`in`** operator checks for existence of a substring

```
pattern in text
```

- The `count` method
    - `text.count(pattern, start, stop)`
    - returns the number of non-overlapping occurrence of `pattern` in `text`
    - `start` and `stop` specify where in the `text` search is to be done
    - `stop` or both `start` and `stop` can be omitted
    - works for strings, list, tuples

Searching for a pattern (string) in a text (string)

- The `find` method
  - `text.find(pattern, start, stop)`
  - returns the starting index of the first from the left occurrence of `pattern` in `text`
  - return -1 if none occurrence was found
  - `start` and `stop` specify where in the `text` search is to be done
  - `stop` or both `start` and `stop` can be omitted
  - works only for strings
- The `index` method
  - as `find`, but
  - if no occurrence of pattern is found raises the `ValueError` exception
  - works also for lists and tuples

Finding multiple copies of the pattern

- Let's complicate a little, now we want to find all occurrences of the given pattern and do something with them
- Here we need the `start` parameter in `find`

```
pos = 0
# Searching for pattern in text[pos:]
while (pos := text.find(pat, pos)) != -1:
    assert text[pos:pos+len(pat)] == pat
    # Here we can process the occurrence
    pos = pos + len(pat)
```

Replacing a pattern (not) in a string

- The `replace` method
- Replacement is a problem, since strings are immutable
- Therefore replace returns a modified copy of the original
    - `text.replace(old, new, count)` ¶
    - gives a copy of the text with the given number of occurrences (from left) of `old` text replaced by the `new` text
    - the `count` parameter may be skipped (which means replacing all occurrences)
- Hence replacing all occurrences of pattern `pat` for a new text `subst` in `text` is easy (but does not change `text`):

```
new_text = text.replace(pat, subst)
```

Replacing a pattern (not) in a string

- Now let's complicate a little bit: let's assume that consecutive instances of the pat have to be replaced with (sometimes) different new texts
- Now we have to find and replace ourselves

```
new_text, old_pos, pos = "", text.find(pat), 0
# text up to old_pos is already copied with replacement
while (pos := text.find(pat, old_pos)) != -1:
    new_text += text[old_pos:pos] + give_subst(pos)
    # give_subst somehow calculates the new text
    old_pos = pos + len(pat)
# do not forget about the tail of the text
new_text += text[old_pos:]
```

Is it enough?

- So far so gut - even with multiple copies and different replacement strings!
- Unfortunately this approach does not work for multiple patterns
- Hence we need something else

Theoretical introduction

- Chomsky hierarchy
- Regular languages
    - Deterministic Finite Automaton (DFA)/Non-deterministic Finite Automaton (NFA)
    - Regular grammar
    - Regular expressions

Formal definition of regular expressions

- For a (finite) set of symbols (alphabet) regular expression is defined as follows
- empty expression (generally denoted by epsilon) is a r.e.
- a single symbol from the alphabet is a r.e.
- If R and S are regular expressions, then
  - RS is a r.e. (concatenation)
  - R | S is a r.e. (alternation)
  - R* is a r.e (repetition, Kleene's star)
- Parenthesis are allowed
- Operator priorities are: *, concatenation, alternation
- Examples:
  - 0 (single symbol "0": {"0"}))
  - 1* (set of finite sequences of "1": {epsilon, "1", "11", "111", ... })
  - 0 | 1 (set of two words "0" and "1": {"0", "1"})
  - (O|1)*0 (set of all even non negative numbers in binary notation with leading zeros: {"0", "00", "10", ...})
  - 0|1(0|1)*0 (set of all even nonnegative numbers in binary notation without leading zeros: {"0", "10", "100", "110", ...})

Notation

- Generally *regex* - regular expression
- Pronunciation dilemma with regexes ('reg-ex' or 'rejex' both are commonly used)
- But regex's in most tools have built in some additional capabilities making them stronger than original regular expressions
- There are many versions of regexes
- Even programming languages provide libraries with different implementations of regexes

What for regexes are useful?

- They can be found in (good) text editors
- Are used in many tools (like grep)
- Are supported by many (most) programming languages
- Allow for easy and quite universal searching/replacing of patterns in texts
- Are highly useful when verifying user entries (like emails, post codes etc.)
- Are extremely useful when extracting data from text files
- they are similar to wildcards (like * and ? in file name searches) but more powerful

The `re` package

- A built-in module (hence no installing hassle)
- With a nicely short name: re (hence no 'import … as …' needed)
- To import write

```
import re
```

- It works with both utf-8 strings and byte arrays
  - But they cannot be mixed in one call to a function from this module

Various operating modes

- Finding all occurrences (matching given regex)
  - `findall`
- Creating a match object for several searches
  - `search`
- Splitting the text at occurrences of the specified regex
  - `split`
- Replacing matches with given string
  - `sub`

Starters (some simple examples)

```
txt = "Python is good, Python is nice, Python rules!"  # no indoctrination intended here

# find all pythons (and possible more due to the wildcard)

lst = re.findall("P....n", txt)  #   ['Python', 'Python', 'Python']

# play with match object

match = re.search("P....n", txt) #   <re.Match object; span=(0, 6), match='Python'>

# split at comma with space

lst = re.split(", ", txt)          # ['Python is good', 'Python is nice', 'Python rules!']

# replace Python with the name of any other of your favorite languages

# (just joking, of course it should be Python)

lst = re.sub("P....n", "C++", txt)  # C++ is good, C++ is nice, C++ rules!
```

The search function

- Syntax

  `search(pattern, text, flags = 0)`

- Searches for the first (from left) match of `pattern` within the `text`
- If successful returns a *match object*, otherwise returns None

The match object

- Represents a match in a text
- Is used for getting deeper information about the match (like position or contents)
- In logical expressions it is interpreted as `True`
  - therefore functions looking for a match (like search or match) return a match or None (which is interpreted in logical expressions as False) what allows for handy use in **if**'s or **while**'s conditions
- When displayed, looks like

  ```
  <re.Match object; span=(7, 26), match=' janusz@mimuw.edu.pl '>
  ```

  and contains
  - position (in range-like notation) within text of the matched substring
  - and the matched substring

The match object

- Has useful methods
- groups
  - returns a tuple with all groups (see later) in the match
- group(id)
  - returns the specified by id group
  - id may be a number (starting from 1(!)) of the group
  - id equal zero means the entire match
  - id may be the name of the group
  - id may take the form of sequence of groups ids
  - id causes an error if there is no corresponding group

How patterns are build?

- Patterns contain
  - plain characters
  - meta-characters
- Plain characters
  - letters, digits, etc.
  - all characters which are not defined as meta-characters
- Meta-characters
  - here is the power of regex expressions
  - they allow for construction of sophisticated expressions
  - are described on consecutive slides

A practical note

- Regexes often use the \ character (see following slides).
- Because of Python's string literals:
  - the backslashes should be doubled (less convenient),
  - or the strings should be preceded with r (raw strings).

Meta-characters: a backslash

- \ backslash
- Many meanings
  - escaping metacharacters
  - special character classes (see later)
- Remember (again) about Python string literal interpretation
  - `"\\\\"` stands for one escaped backslash
  - `r"\\"` as above
- Simple rule: always use in Python raw strings as regexes (just in case)

Meta-characters: a pair of brackets

- [] (square) brackets
- Stands for one character to be matched in the text
- Contains specification of a group (mostly) of characters
- These characters may be just listed (without spaces or commas)
    - `[abc]` stands for a or b or c
- They may be specified as a range
    - `[a-z]` any lowercase letter (from English alphabet)
- The matching character are also called a *character class*

Meta-characters: a dot

- . dot
- stands for any character to be matched in the text (with the exception of a newline)
- Example

    ... any three characters (except for newlines)

Meta-characters: a circumflex

- ^ circumflex
- Has two meanings
- Alone represents (anchors the match at) the start of the string

  `^Susan..` the word Susan at the beginning of a string followed by any two characters (not being new lines)

- \A works the same (apart from multiline mode)
- If at the start within brackets negates its meaning, i.e. makes the expression represent complement of the character class

  `[^a-z]` any character but not a lowercase letter (from English alphabet)

Meta-characters within []

- ^ has special meaning only at the first position

  `[^]` error

  `[a^1]` a or ^ or 1 (here ^ has no special meaning)

  `[\^x]` ^ or x (^ is escaped here)

- - has special meaning but not as the first or the last character

  `[-1]` - or 1

  `[a-]` a or -

  `[a\-z]` a or - or z

  `[z-a]` error

Meta-characters within []

-] has special meaning with the exception to the first position

  `[]a]`] or a

  `[a\]]` a or]

- other meta-character lose their special meaning within []

  `[#*+.]` # or * or + or .

- but special character classes (see later) work fine within []

  `[\d\s]` # decimal digit or whitespace

Meta-characters: a dollar

- $ dollar
- Just anchors the pattern at the end of the text
- \Z works the same (apart from multiline mode)
- Example:

`Rosemary$` the name must end the text

Meta-characters for repetitions

- `*` (asterisk) - zero or more repetitions
- `+` (plus) - one or more repetitions
- `?` (question mark) - zero or one repetition (has also other meanings)
- `{m,n}` - number of repetitions must be in the specified range m...n (n is included)
  - m or n can be skipped, m defaults to 0, n to any (like asterisk)
  - if one is skipped (like in `{m}`) the number specifies `{m,m}` (i.e. the exact number of repetitions)
  - note that `{m,}` is different form `{m}` and both are different from `{,m}`
  - `{}` means just a pair of curly brackets (literally), whereas `{,}` means any number of repetitions
- Examples:

`[a-z]*` any (including empty) sequence of lowercase letters (of English alphabet)

`[a-z]+` any nonempty sequence of lowercase letters (of English alphabet)

`[a-z]?` one or none lowercase letter (of English alphabet)

`[a-z]{2,3}` two or three lowercase letters (of English alphabet)

Meta-characters: backslash

- Has three meanings:
  - escapes a meta-character
  - starts the name of a special character class (those are given later)
  - references a previous group (very powerful option)!
- Remember about r before the pattern string!
- Examples:

`\.\*`	a dot followed by an asterisk
`\d`	a (decimal) digit
`(a)\1`	two letters a

Metacharacters: alternative

- | vertical bar (aka pipe)
- Denotes alternative (like in regular expressions)
- Has lower priority then concatenation (which is weaker than *)
- Easy to understand but makes matching more expensive (backtracking)
- Example

  `a|b` letter a or letter b

Meta-characters: creating groups

- `()` parenthesis
- Represents a substring of the matched string
- Enables specification of patterns beyond the regular expressions expressive power
- Very handy when analyzing matched portions of text
- Some editors enable using it in the replace command
  - for example: find `(\[a-z])\d\d(\[a-z])` and replace with `\2##\1` replace two digits numbers by ## only when they surrounded by letters, revert order of these two letters
- Example:

  `([a-z])[a-z]*\1` word (lowercase English letters) with the same start as end

Meta-characters: named groups

- Like normal groups but with `?P<name>` after the left parenthesis
- They are also counted as regular (normal) groups
- The `name` must be a valid Python identifier
- Are referenced by (?P=name) (or \number)
- Example:

```
(?P<start>[a-z])[a-z]*(?P=start)
```

```
(?P<start>[a-z])[a-z]*\1
```

in both cases word (lowercase English letters) with the same start as its end

Special characters classes

- `\d` any decimal digit
- `\D` any but decimal digit character
- `\w` alphanumeric character
  - digits and unicode letters (for string, i.e. unicode regexes)
  - `[a-zA-Z0-9_]` with ASCII flag or for byte regexes
  - for byte regexes alphanumeric characters in current locale and underscore if LOCALE flag is set
- `\W` opposite of `\w`
  - (e.g. with the ASCII flag any but alphanumeric character i.e. `[^a-zA-Z0-9_]`)
- `\s` any whitespace character (including \n!)
- `\S` any but whitespace character (space, tab, newline)

Anchors

- We have learn some already
- They do not consume characters from the text
- ^ and /A start of the text
- $ and /Z end of the text
  - $ (not /Z) matches also before one \n at the end of the text i.e. `no$` matches `anno`, `anno\n` but not `anno\n\n`
- /b word boundaries (a word is understood as \w+)
  - `\bpro\w*` matches `programmer` but not `unproductive`
  - `\band\b` matches `and` but not `pandas`
  - `able\b` matches words ending with able
- /B not at word boundary

How far * extends?

- Two modes
  - regular (greedy) - longest possible match
  - lazy (non-greedy) - shortest possible match
- Non-greedy versions of `*`,`+`,`?`
  - `*?`
  - `+?`
  - `??`
  - `{m,n}?`
- Examples
  - `.*` in `one\ntwo\nthree` matches only `one` (hence not so greedy)
  - `<.*>` in `<b>page</b>` matches entire text (yet greedy)
  - `<.*?>` in `<b>page</b>` matches only `<b>`
  - `<[^>]*>` would work as well

More

- We covered here the most important part of regexes
- There is still more to regexes in Python

Thank you for your attention!