Strings

Regular expressions

Operations on strings

- In addition to strings allowing indexing, concatenation, slicing, and repetition operations, string class also offers many methods
- Since strings in Python are immutable, these methods work by returning a value. The methods leave the object pointed by `self` intact
- In the following slides several of these methods are briefly described. The methods are divided into five groups.

Classification of characters (and strings)

- All the following methods will take one string as parameter. An empty string will always result a False return value.

`isalnum`	True if all characters are letters or digits
`isalpha`	True if all characters are letters
`isdigit`	True if all characters are digits
`islower`	True if contains letters, and all are lowercase
`isupper`	True if contains letters, and all are uppercase
`isspace`	True if all characters are whitespace
`istitle`	True if uppercase in the beginning of word, elsewhere lowercase

String transformations

- The following methods do conversions between lower- and uppercase characters in the string. All these methods return a new string.

`lower`	Change all letters to lowercase
`upper`	Change all letters to uppercase
`capitalize`	Change to capitalcase
`title`	Change to titlecase
`swapcase`	Change all uppercase letters to lowercase, and vice versa

- The method `translate(`*`table, deletechars`*`='')` will transform all characters using the string *table* that has length 256. All characters appearing in *deletechars* will be deleted.

Searching for substrings

- All the following methods get the wanted substring as the parameter, except the replace method, which also gets the replacing string as a parameter

`count`	Counts the number of occurences of a substring
`find`	Finds index of the first occurence of a substring, or -1
`rfind`	Finds index of the last occurence of a substring, or -1
`index`	Like `find`, except `ValueError` is raised if not found
`rindex`	Like `rfind`, except `ValueError` is raised if not found
`startswith`	Returns `True` if string starts with a given substring
`endswith`	Returns `True` if string ends with a given substring
`replace`	Returns a string where occurences of one string are replaced by another

Trimming and adjusting

`strip(x)`	Removes leading and trailing whitespace, or characters found in string `x`
`lstrip(x)`	Same as strip but only leading characters are removed
rstrip(x)	Same as strip but only trailing characters are removed
`ljust(n)`	Left justifies string inside a field of length `n`
`rjust(n)`	Right justifies string inside a field of length `n`
`center(n)`	Centers string inside a field of length `n`
`expandtabs(tabsize)`	Expands all tab characters into spaces, with tabstop at every *tabsize* characters

Joining and splitting

- `join(seq)` joins a sequence *seq* of strings. The string itself is used as a delimitter. An example: `"XY".join(["abc", "def", "ghi"])` results the string `'abcXYdefXYghi'`
- `split(sep=None)` divides a string into pieces that are separated by the string *sep*. The pieces are returned in a list. For instance, the call `'abcXYdefXYghi'.split('XY')` will result in `['abc', 'def', 'ghi']`
- `splitlines` is like `split('\n')`

The string module 1

- The `string` module contains functions that duplicate the methods listed on the previous few slides
- Sometimes the function form looks more natural than the method form, and vice versa
- The `string` module also contains several data attributes:
  - `string.digits` == '0123456789'
  - `string.punctuation` == `"!"#$%&\'()*+,-./:;<=>?@[]^_{|}~"`
  - `string.whitespace` is a string with the following special characters: space, tab, linefeed, and carriage return

The string module 2

- String module data attributes continues:
  - `string.ascii_lowercase` == "abcdefghijklmnopqrstuvwxyz"
  - `string.ascii_uppercase` == "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
  - `string.ascii_letters == ascii_lowercase + ascii_uppercase`
- The following attributes are `locale` (*local environment*) dependent: letters, lowercase, uppercase
- For example, in the Finnish locale setting the letters would also include ä and ö

The textwrap module

- The `textwrap` module includes the following two functions for text formatting, among others:
- The function `wrap(s, width = 70)` will return a list of strings whose length is at most *width*. The catenation of these strings would more or less be the original string `s`.
- The function `fill(s, width=70)` returns a string where it has inserted newline characters (\n) so that there is a line break after at least every *width* characters.
- Example:

```
>>> textwrap.wrap(string.lowercase, 10)
['abcdefghij', 'klmnopqrst', 'uvwxyz']
>>> textwrap.fill(string.lowercase, 10)
'abcdefghij\nklmnopqrst\nuvwxyz'
```

Examples of regular expressions 1

- We have already seen that we can ask from a string `str` whether it begins with some substring as follows: `str.startswith('Apple')`
- If we would like to know whether it starts with "Apple" or "apple", we would have to call `startswith` method twice
- *Regular expressions* offer a simpler solution: `re.match(r'[Aa]pple', str)`
- The bracket notation is one example of the special syntax of regular expressions. In this case it says that any of the characters inside brackets will do: either 'A' or 'a'. The other letters in "pple" will act normally. The string `r'[Aa]pple'` is called a *pattern*.

Examples of regular expressions 2

- A more complicated example asks whether the string `str` starts with either apple or banana (no matter if the first letter is capital or not):
  `re.match(r'[Aa]pple|[Bb]anana', str)`
- In this example we saw a new special character | that denotes an alternative. On either side of the bar character we have a *subpattern*.

Examples of regular expressions 3

- A legal variable name in Python starts with a letter or an underline character and the following characters can also be digits
- So legal names are, for instance: `_hidden`, `L_value`, `A123_`. But the name `2abc` is not a valid variable name.
- Let's see what would be the regular expression pattern to recognise valid variable names:

```
r'[A-Za-z_][A-Za-z_0-9]*\Z'
```

- Here we have used a shorthand for character ranges: A-Z. This means all the characters from A to Z.

Examples of regular expressions 4

- The first character of the variable name is defined in the first brackets. The subsequent characters are defined in the second brackets.
- The special character * means that we allow any number (0,1,2, ...) of the previous subpattern. For example the pattern `r'ba*'` allows strings `'b'`, `'ba'`, `'baa'`, `'baaa'`, and so on.
- The special syntax `\Z` denotes the end of the string
- Without it we would also accept `abc$` as a valid name since `match` normally checks only that a string starts with a pattern.

Examples of regular expressions 5

- The special notations, like `\Z`, also cause problems with string handling.
- Remember that normally in string literals we have some special notation: `\n` stands for newline, `\t` stands for tab, and so on.
- So, both string literals and regular expressions use similar looking notations, which can create serious confusion.
- This can be solved by using the so-called *raw strings*. We denote a raw string by having an `r` letter before the first quotation mark, for example `r'ab*\Z'`
- When using raw strings, the newline (`\n`), tab (`\t`), and other special string literal notations aren't interpreted. One should always use raw strings when defining regular expression patterns.

What do patterns do?

- A pattern represents a set of strings. This set can even be potentially infinite.
- They can be used to describe a set of strings that have some commonality; some regular structure
- Regular expressions are a classical computer science topic
- They are very common in programming tasks. Scripting languages, like Python, are very fluent in regular expressions.
- Very complex text processing can be achieved using regular expressions

RE pattern syntax

- Normal characters (letters, numbers) just represent themselves, unless preceded by a backslash, which may trigger some special meaning
- Punctuation characters have special meaning, unless preceded by backslash (\), which deprives their special meaning
- Use \\to represent a backslash character without any special meaning
- In the following slides we will go through some of the more common RE notations

Wildcards, anchors, and repetitions

.	Matches any character
[...]	Matches any character contained within the brackets
[^...]	Matches any character not appearing after the hat (^)
^	Matches the start of the string
$	Matches the end of the string
*	Matches zero or more previous RE
+	Matches one or more previous RE
{m,n}	Matches m to n occurences of previous RE
?	Matches zero or one occurences of previous RE

Alternatives, groupings, and backreferences

- We have already seen that a `|` character denotes alternatives. For example, the pattern `r'Get (on|off|ready)'` matches the following strings: "Get on", "Get off", "Get ready"
- We can use parentheses to create groupings inside a pattern: `r'(ab)+'` will match the strings "ab", "abab", "ababab", and so on
- These groups are also given a reference number starting from 1. We can refer to groups using backreferences: *number*
- For example, we can find separated patterns that get repeated: `r'([a-z]{3,}) \1 \1'`
- This will recognise, for example, the following strings: `"aca aca aca"`, `"turn turn turn"`. But not the strings `"aca aba aca"` or `"ac ac ac"`.

Shorthands for character sets

- In the following, note that a hat (^) as the first character inside brackets will create a complement set of characters

`\d`	same as [0-9], matches a digit
`\D`	same as [^0-9], matches anything but a digit
`\s`	matches a whitespace character (newline, tab, ...)
`\S`	matches a nonwhitespace character
`\w`	same as [a-zA-Z0-9_], matches one alphanumeric character
`\W`	matches one non-alphanumeric character

- Using the above notation we can now shorten our previous variable name example to `r'[a-zA-Z_]\w*\Z'`

Breaks

- The patterns \A, \b, \B, and \Z will all match an empty string, but in specific places
- The patterns \A and \Z will recognise the beginning and end of the string, respectively
- Note that the patterns ^ and $ can in some cases match also after a newline and before a newline, correspondingly.
- So, \A is distinct from ^, and \Z is distinct from $
- The pattern \b matches at the start or end of a word. The pattern \B does the reverse.

match and search functions

- We have so far only used the `re.match` function which tries to find a match at the beginning of a string
- The function `re.search` allows to match any substring of a string
- Example: `re.search(r'\bback\b', str)` will match strings "back", "a back, is a body part", "get back". But it will not match the strings "backspace" or "comeback".

Examples 1

- The function `re.search` finds only the first occurence.
- We can use the `re.findall` function to find all occurences.
- Let's say we want to find all *present participle* words in a string *str*. The present participle words have ending 'ing'
- The function call would look like this:
  `re.findall(r'\w+ing\b', str)`
- Let's try running this:

```
>>> str = "Doing things, going home, staying awake,
           sleeping later"
>>> re.findall(r'\w+ing\b', str)
['Doing', 'going', 'staying', 'sleeping']
```

Examples 2

- Let's say we want to pick up all the integers from a string.
- We can try that with the following function call:
  `re.findall(r'[+-]?\d+', str)`
- An example run:

```
>>> re.findall(r'[+-]?\d+', "23 + -24 = -1")
['23', '-24', '-1']
```

Examples 3

- Suppose we are given a string of if/then sentences, and we would like to extract the conditions from these sentences.
- Let's try the following function call:
  `re.findall(r'[Ii]f (.*), then', str)`
- An example run:

```
>>> str=("If I'm not in a hurry, then I should stay. " +
     "On the other hand, if I leave, then I can sleep.")
>>> re.findall(r'[Ii]f (.*), then', str)
["I'm not in a hurry, then I should stay.
 On the other hand, if I leave"]
```

- But I wanted a result: ["I'm not in a hurry", 'I leave']. That is, the condition from both sentences. How can this be fixed?