Examples 4

- The problem is that the pattern `.*` tries to match as many characters as possible.
- This is called *greedy* matching
- One way of solving this problem is to notice that the two sentences are separated by a full-stop (.)
- So, instead of matching all the characters, we need to match everything but the dot character
- This can be achieved by using the complement character class: `[^.]`. The hat character (^) in the beginning of a character class means the complement character class

Examples 5

- After the modification the function call looks like this:

```
re.findall(r'[Ii]f ([^.]*), then', str)
```

- Another way of solving this problem is to use a *non-greedy* matching
- The repetition specifiers `+`, `*`, `?`, and `{m,n}` have corresponding non-greedy versions: `+?`, `*?`, `??`, and `{m,n}?`
- These expressions use as few characters as possible to make the whole pattern match some substring
- By using non-greedy version, the function call looks like this: `re.findall(r'[Ii]f (.*?), then', str)`

Functions in the re module 1

Below is a list of the most common functions in the re module

- re.match(pattern, str)
- re.search(pattern, str)
- re.findall(pattern, str)
- re.finditer(pattern, str)
- re.sub(pattern, replacement, str, count=0)

Functions in the `re` module 2

- Functions `match` and `search` return a *match object*
- A match object describes the found occurence
- The function `findall` returns a list of all the occurences of the pattern. The elements in the list are strings
- The function `finditer` works like `findall` function except that instead of returning a list, it returns an iterator whose items are match objects
- The function `sub` replaces all the occurences of the pattern in *str* with the string *replacement* and returns the new string

The sub function 1

- An example: The following program will replace all "she" words with "he"

```
import re
str = "She goes where she wants to, she's a sheriff."
newstr = re.sub(r'\b[Ss]he\b', 'he', str)
print newstr
```

- This will print

```
he goes where he wants to, he's a sheriff.
```

The sub function 2

- The sub function can also use backreferences to refer to the matched string. The backreferences \1, \2, and so on, refer to the groups of the pattern, in order
- An example:

```
import re
str = """He is the prime minister of Russia.
He's a powerful man."""
newstr = re.sub(r'(\b[Hh]e\b)', r'\1 (Putin)', str, 1)
print newstr
```

- This will print

```
He (Putin) is the prime minister of Russia.
He's a powerful man.
```

The match object 1

- Functions `match`, `search`, and `finditer` use match objects to describe the found occurence
- The method `groups()` of the match object returns the tuple of all the substrings matched by the groups of the pattern
- Each pair of parentheses in the pattern creates a new group
- These groups are are referred to by indices 1, 2, ...
- The group 0 is a special one: it refers to the match created by the whole pattern

The match object 2

- Let's look at the match object returned by the call

```
mo = re.search(r'\d+ (\d+) \d+ (\d+)',
               'first 123 45 67 890 last')
```

- The call `mo.groups()` returns a tuple `('45', '890')`
- We can access just some individual groups by using the method `group(gid, ...)`
- For example, the call `mo.group(1)` will return '45'
- The zeroth group will represent the whole match: '123 45 67 890'

The match object 3

- In addition to accessing the strings matched by the pattern and its groups the corresponding indices of the original string can be accessed:
- The `start(gid=0)` and `end(gid=0)` methods return the start and end indices of the matched group gid, correspondingly
- The method `span(gid)` just returns the pair of these start and end indices

The match object 4

- The match object `mo` can also be used like a boolean value:

```
mo = re.search(...)
if mo:
    # do something
```

  will do something if a match was found
- Alternatively, the match object can be converted to a boolean value by the call `found = bool(mo)`

Precompiling the regular expression patterns

- If the same pattern is used in many function calls, it may be wise to precompile the pattern, mainly for efficiency reasons
- This can be done using the compile(pattern, flags=0) function in the `re` module. The function returns a so-called *RE object*
- The RE object has method versions of the functions found in module `re`.
- The only difference is that the first parameter is not the pattern since the precompiled pattern is stored in the RE object.

Optional flags

- The details of matching operation can be specified using optional *flags*
- These flags can be given either inside the pattern or as a parameter to the `compile` function
- Some of the more common flags are given in the following table

`(?i)`	`re.IGNORECASE`
`(?m)`	`re.MULTILINE`
`(?s)`	`re.DOTALL`

- The elements on the left can appear anywhere in the pattern but preferably in the beginning
- On the right there are attributes of the `re` module that can be given to the `compile` function as the second parameter

Optional flags

- The IGNORECASE flag makes lower- and uppercase characters appear as equal
- The MULTILINE flag makes the special characters ^ and $ match the beginning and end of each line in addition to the beginning and end of the whole string. These flags make \A differ from ^, and \Z differ from $
- The DOTALL flag makes the character class . (dot) also accept the newline character, in addition to all the other letters

Optional flags

- When giving multiple flags to the `compile` function, the flags can be separated with the | sign
- For example, `re.compile(pattern, re.MULTILINE | re.DOTALL)`
- This is equal to re.compile('(?m)(?s)' + pattern)