Regular expressions

Script Langugages (INZ002025)

Wojciech Thomas

Department of Applied Informatics
Faculty of Computer Science and Management

13th Nov 2020

European Funds Knowledge Education Development
Wrocław University of Science and Technology
European Union European Social Fund

This week

We are going to learn about:

- raising exceptions
- searching and replacing strings
- regular expressions

Exceptions revisited

Exceptions

- Catching exceptions:

```python
try:
  s = input("Enter a number ")
  i = int(s.strip())
except ValueError:
  print("Could not convert data to an integer.")
else:
  print("Entered value is {}".format(i))
```

Finalizing Exception Handling

- `finally` section is **always** executed as the last operation of the `try` statement:

```python
try:
  s = input("Enter a number ")
  i = int(s.strip())
except ValueError:
  print("Could not convert data to an integer.")
else:
  print("Entered value is {}".format(i))
finally:
  print("User entered: '{}'".format(s)))
```

- Beware: if `try` and `finally` blocks include `return`, value from `finally` block is returned

Raising exceptions

- Exceptions can be risen:
  - automatically
  - by the programmer
- If you:
  - want to show that something is wrong
  - but cannot or do not want to resolve the problem

  You **raise** the exception:

```python
s = input("Enter a number")
i = int(s.strip())
if i <= 0:
  raise ValueError("i should be nonnegative")
```

Raising exception

- You can add the argument to describe the problem in details:

```python
raise ValueError

raise ValueError("i is nonnegative")
```

Advantages of raising exceptions

- You don't have to bother:
  - **how** to handle a problem
  - **where** the problem is handled
  - you can pass additional parameters to help recover the exception or find the culprit

Re-rising Exceptions (1)

- If you enter `except` block, Python considers the problem as solved and continues execution
- If you want your program to stop you must do it explicitly:

```python
import sys

try:
  s = input("Enter a number ")
  i = int(s.strip())
except ValueError:
  print("Could not convert data to an integer.")
  sys.exit()
```

- If you do not want your application to stop, but the problem is not resolved you should raise the exception again

Re-rising Exceptions (2)

- If you catch the exception, however cannot resolve the problem, you can re-raise exception:

```python
try:
  s = input("Enter a number ")
  i = int(s.strip())
except ValueError:
  print("Could not convert data to an integer.")
  raise
```

- If you omit the exception name, the same exception is risen

Re-rising Exceptions (3)

- You could re-raise exception to add some details:

```python
try:
  s = input("Enter a number ")
  i = int(s.strip())
except ValueError:
  raise ValueError("Could not convert '{}' "
                   "to an integer.".format(s))
```

- You could raise outright different exception:

```python
try:
  # some code
except FileNotFoundError:
  raise ValueError("Wrong filename")
```

Exceptions How-to

- It is expensive to handle the exception
- Exceptions should be used when you do not expect to problem
- If the condition is obvious use `if` instead of `try`
- If the condition is exception, use `try`
- If you don’t know how to handle the problem use `raise`

Searching and Replacing Texts

Simple text searching

- If you want to check existence use `in`:

```python
"User" in "User experience"
```

- If you want to get the position of the substring use method `find`

```python
s = "User experience"
pos = s.find("User")
```

`-1` means not found

Simple text replacing

- If you want to replace string use `replace`

```python
s = "User experience"
t = s.replace("User", "Programmer")
```

`replace` does not modify original string

Regular Expressions

Regular Expressions

- Named: RE, regex
- Small and specialized programming language
- You can use it to:
  - check if string matches pattern
  - replace substring in string
  - split string

A History about Regular Expressions

Once upon a time. . .

Once a programmer had a problem.

He decided to solve it using regular expressions.

Now he has got two problems.

Exact matches

- Sequences of characters matches only the same string
- `A` -> “A”
- `AA` -> “AA”
- `ABA` -> “ABA”
- `mp3` -> “mp3”

Repetitions (1)

- `A+` -> “A” “AA” “AAA” etc
- `A*` -> "" “A” “AA” etc.
- `AB+` -> “AB” “ABB” “ABBB” etc.
- `AB*` -> ?
- `A?` -> "" or “A” only
- `+` one-or-more times
- `*` zero-or-more times
- `?` zero-or-one times

Repetitions (2)

- `A{1,2}` -> “A” or “AA” only
- `{m,n}` between m and n
- `+` means `{1,}`
- `*` means `{0,}`
- `?` means `{0,1}`

Grops of characters

- Operators such as `*` regards only single character
- Longer string should be grouped
- `(ABC)+` -> “ABC”, “ABCABC”, “ABCABCABC” etc.
- `(A|B)+` - or - “A” “B” “AAAB” “AABB” etc.
- `(Ana|Eva)` - “Ana” or “Eva” only

Special characters

- `[ABC]+` - one of - “AABCC” “ABBBC” “BBC” etc.
- `[^AB]+` - all characters except “A” and “B”
- `[a-zA-Z0-9_]+` (shortly: `\w`) – a word
- `[0-9]` (shortly: `\d`) – a decimal digit
- `[ \t\n\r\f\v]` (shortly: `\s`) – a whitespace
- `.*` – any string
- `.` – any character except `\n` (NEWLINE)
- If you want to use any of the special characters, eg. `{ } [ ] ( ) . + * \ ^ $` it must be preceded by `\` (Escape character)

Lines

- `^` (caret) - start of the line
- `^abc.*` – will match all lines starting with `abc`
- `$` – end of the line
- `.*k$` – will match all lines ending with `k`

Escape character

- RE special characters must be prepended with escape character `\`
- So...
  - For regex to find `\chapter` string you should use `\\chapter`
- But...
  - Python string requires `\` to be prepended with `\`
- So...
  - We get: `"\\\\chapter"` in Python
- Solution: use Python raw string: `r"\\chapter"`
  - `r` before string means Python will not interpret content of the string

Greedy match

- RE by default uses **greedy** approach
  - try to match as many character as it is possible
- Example:
  - regular expression `"<.*>"` used against `"<h1>Header</h1>"` will match the **whole** string and not `"<h1>"` only

Compiling expressions

- Regular Expressions are supported by `re` module
- Before use, RE expressions must be compiled

```
import re
p = re.compile("a[bcd]*d")
p = re.compile("a[bcd]*d", re.IGNORECASE)
```

- `compile` method support parameters:

```
p = re.compile("a[bcd]*d", re.IGNORECASE)
```

Matching a whole expression

- Finding a match:

```
p = re.compile("ab*")
m = p.match("abbb")
```

- Match object contains:
  - group - matching pattern
  - start
  - end
  - span - tuple (start,end)

Matching examples

- Checking if a phone number has a proper format:
  `\(\d{2}\) \d{3} \d{2} \d{2}`

  Matches “(71) 320 12 34”
- Checking if a string is a valid e-mail address:
  `\w@([-a-z0-9]+\.)+[-a-z0-9]{2,}`
  - DNS domain can contain only characters `a-z`, digits `0-9` and a dash `-`
- Checking if a string is a numeric IP address:
  `\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}`

  Beware: it **does not** check validity: e.g. 999.999.999.999 matches an expression, but is an invalid IP address

Using Groups

- You can use groups (limited by `()`) to find elements of the matched string

```
p = re.compile("([A-Z][a-z]{2}) "
               "(\d{1,2})[a-z]{0,2}, (\d{4})")
l = p.match("Jan 1st, 1970")
```

Result:

- `group(0)` matches the whole string
- the subsequent groups are numbered by starting bracket `(` `group(1)` -> “Jan”, `group(1)` -> “1”, `group(3)` -> “1970”
- groups can be nested

Finding multiple matches (1)

- Find all matches of expression:

```
p = re.compile("ab*")

l = p.findall("a abbb abc  dac")
```

- Result is the list of matched expressions:

```
["a", "abbb", "ab", "a"]
```

Finding multiple results (2)

- Get all matches as an iterator

```
p = re.compile("ab*")

iter = p.finditer("ac abbb abc d")
for match in iter:
  print(match.group())
```

Module functions

- Methods of regular expression object are available as a functions:

```python
import re
m = re.match("ab*", "abbb")
```

- There are functions such as `match`, `search`, `findall` that works as corresponding `re` methods
  - Return `None` if there is no match
  - Return corresponding object (eg. match or iterator) in case of match
  - Each function internally calls `re.compile()` and then the corresponding method

Thank you for you attention

- See you next week