Modules and Regular Expressions

Info 206

Niall Keleher

26 September 2017

Today's Outline

1. Module Packages
2. Regular Expressions
3. Exercise - Regular Expressions

Module Packages

Namespace Packages

- Allow packages to span multiple directories
- Get rid of the need for multiple `__init__.py` files (Python 3.2 and earlier required the `__init__.py` file in the directory listed in the `import` or `from` statement)
- Allows for more efficient and logical package look up

Module design concepts

- `__name__` is used to set and test the namespace of the module. (Good to include in unit tests.)
- `__main__` applies when the module is run, not when it is imported
- Docstrings - Use them and provided detailed information for modules and functions

Regular Expressions

```
import re
source = "To be or not to be, that is a question"
pattern = "To be"

result = re.match(pattern, source)
print(result)
```

You can precompile a regular expression to speed up the query.

```
compiled_pattern = re.compile("To be")
```

You can precompile a regular expression to speed up the query.

```
compiled_pattern = re.compile("To be")
```

You can then use the compiled pattern to execute the match query.

```
result_compiled = compiled_pattern.match(source)
print(result_compiled)
```

There are a few ways to use the re package to find patterns in strings:

- `search()`: returns the first match

- `findall()`: returns a list of all nonoverlapping matches

- `split()`: returns a list of strings that were split from the source string at the locations where the pattern was found

- `sub()`: returns a string that has replaced all instances of the pattern with a replacement string passed into the function

Exact match with `match()`

```
import re
source = "To be or not to be, that is the question"

# We are creating a compiled pattern that we can
# use in multiple searches
compiled_pattern = re.compile("To be")

# Now we are searching to see if the
# compiled_pattern is in the front of the source text
m = compiled_pattern.match(source)

if m: # if there is a match
    print(m.group()) # let's print out what was matched
```

What would happen if we tried to match using the pattern "that is"?

```
middle_pattern = re.compile("that is")
m = middle_pattern.match(source)

if m:
    print(m.group())
```

```
middle_pattern_with_wildcard = re.compile(".*that is")
m = middle_pattern_with_wildcard.match(source)
if m:
    print(m.group())
```

- "." matches any single character
- "*" matches any number of the previous character
- So the combination ".*" means any number of any character (even zero)
- "that is" is the string we would like to match

Match with `search()`

```
middle_pattern = re.compile("that is")
m = middle_pattern.search("that is")

if m:
    print(m.group())
```

All matches with `findall()`

```
#Find all of the n's in the source string
n_pattern = re.compile("n")
m = n_pattern.findall(source)
print("Found", len(m), "matches")
print(m)
```

Special characters

The `re` package provides a set of character sequences that begin with a backslash for use in regular expressions. Each of these matches a common set of useful characters.

Pattern	Matches
\d	a single digit
\D	a single non-digit
\w	an alphanumeric character
\W	a non-alphanumeric character
\s	a whitespace character
\S	a non-whitespace character
\b	a word boundary (between a \w and a \W, in either order)
\B	a non-word boundary

Example of special characters

```
import string
printable = string.printable
print(printable)
```

```
import re
re.findall("\d", printable)
```

```
printable = string.printable
re.findall("\w", printable)
```

```
re.findall("\s", printable)
```

Pattern	Matches
abc	literal abc
(expr)	expr
expr1 | expr2	expr1 or expr2
.	any character except \n
^	start of source string
$	end of source string
prev ?	zero or one prev
prev *	zero or more prev, as many as possible
prev *?	zero or more prev, as few as possible
prev +	one or more prev, as many as possible
prev +?	one or more prev, as few as possible
prev { m }	m consecutive prev
prev { m, n }	m to n consecutive prev, as many as possible
prev { m, n }?	m to n consecutive prev, as few as possible
[abc]	a or b or c (same as a|b|c)
[^ abc]	not (a or b or c)
prev (?= next)	prev if followed by next

A more realistic example

```
large_source = """
Hi Bianca,
It was great to talk to you about regular expressions. I really under
them more than I ever had before. Would you like to work on the next
together? My number is 650-555-3948. Thanks and talk to you soon!

-Mary
"""
```

What we want to do is build a pattern that would do the following:

```
match three numbers
followed by a dash
then match three more numbers
followed by a dash
and then match four numbers
```

What we want to do is build a pattern that would do the following:

```
match three numbers
followed by a dash
then match three more numbers
followed by a dash
and then match four numbers
```

```
phone_number_pattern = re.compile(r'[0123456789]{3}-[0123456789]{3}-[
m = phone_number_pattern.findall(large_source)
print(m)
```

We could have also written this regular expression more compactly as follows:

```
phone_number_pattern = re.compile(r'\d{3}-\d{3}-\d{4}')
m = phone_number_pattern.findall(large_source)
print(m)
```

let's improve our phone number regular expression by providing the ability to grab the area code and the rest of the number.

```
phone_number_pattern = re.compile(r'(\d{3})-(\d{3}-\d{4})')
m = phone_number_pattern.search(large_source)

if m:
    print(m.group())
    print(m.groups())
```

You can also name the groups for easy retrieval.

```
import re
phone_number_pattern = re.compile(r'(?P<areacode>\d{3})-(?P<number>\d
m = phone_number_pattern.search(large_source)

if m:
    print(m.group("areacode"))
    print(m.group("number"))
```

Python documentation for `re`

https://docs.python.org/3/library/re.html#module-re

Testing Regular Expressions

http://pythex.org

Exercise

End of Meeting #10

For next meeting

- Videos:

    1. Object-Oriented Programming (5 mins)
    2. Classes Introduction (3 mins)
    3. Classes and Attributes (12 mins)
    4. Using the Class Definition (12 mins)
    5. Binding Methods (2 mins)
    6. Initializing a Class (15 mins)
    7. Controlling Access to Attributes (17 mins)
    8. Class Odds and Ends (18 mins)

- Readings:

    - Lutz Chapter 26: OOP: The Big Picture
    - Lutz Chapter 27: Class Coding Basics