CNT 4603: System Administration
Spring 2013

Python – Part 2

Instructor : Dr. Mark Llewellyn
markl@cs.ucf.edu
HEC 236, 4078-823-2790
http://www.cs.ucf.edu/courses/cnt4603spr2013

Department of Electrical Engineering and Computer Science
Computer Science Division
University of Central Florida

Regular Expressions In Python

- Regular expressions (regex) are one of the black arts of practical modern programming. Those who master regular expressions will find that they can solve many problems quite easily while those who don't will waste many hours pursuing complicated work-arounds.

- Regular expressions, although complicated, are not really difficult to understand. Fundamentally, they are a way to describe *patterns* of text using a single set of strings.

- Unlike a simple search-and-replace operations, such as changing all instances of "`Marty`" to "`Mark`", regex allow for much more flexibility – for example, finding all occurrences of the letters "`Mar`" followed by either "`ty`" or "`k`", and so on.

Regular Expressions In Python

- Regular expressions were initially described in the 1950s by a mathematician named S.C. Kleene, who formalized models that were first designed by Warren McCulloch and Walter Pitts to describe the human nervous system.
- Regex were not actually applied to computer science until Ken Thompson (one of the original designers of the Unix OS) used then as a means to search and replace text in his *qed* editor.
- Regex eventually made their way into the Unix operating system (and later into the POSIX standard) and into Perl as well, where they are considered one of the language's strongest features.

Regular Expressions In Python

- Python supports the Perl5 standard of regular expressions.
- The Perl version is known as PCRE (Perl-Compatible Regular Expressions).
- PCRE are much more powerful than their POSIX counterparts – and consequently more complex and difficult to use.

Regular Expressions In Python

- Regex is, essentially, a whole new language, with its own rules, own structures, and its own quirks. What you know about other programming languages has little or no bearing on regex, for the simple reason that regular expression are highly specialized and follow their own rules.

Regular Expression Axioms as defined by S. C. Kleene

- A single character is a regular expression denoting itself.
- A sequence of regular expressions is a regular expression.
- Any regular expression followed by a * character (known as the "Kleene Star") is a regular expression composed of zero or more instances of that regular expression.
- Any pair of regular expressions separated by a pipe character (|) is a regular expression composed of either the left or the right regular expression.
- Parentheses can be used to group regular expressions.

Regular Expressions In Python

- While Kleene's definition of what makes a regular expression might, at first, seem confusing, the basics are actually pretty easy to understand.
- First, the simplest regular expression is a single character. For example, the regex `a` would match the character "a" in the word "Mark".
- Next, single character regex can be grouped by placing them next to each other. Thus the regex `Mark` would match the word "Mark" in "Your instructor is Mark for CNT 4603."
- So far, regex are not very different from normal search operations. However, this is where their similarities end.

Regular Expressions In Python

- The Kleene Star can be used to create regex that can be repeated any number of times (including none).
- Consider the following string:

```
seeking the treasures of the sea
```

- The regex `se*` will be interpreted as "the letter s followed by zero or more instances of the letter e" and will match the following:
  - The letters "see" of the work "seeking", where the regex `e` is repeated twice.
  - Both instances of the letter `s` in "treasures", where `s` is followed by zero instances of `e`.
  - The letters "se" of the work "sea", where the `e` is present once.

Regular Expressions In Python

- It's important to understand in the regex `se*` that only the expression e is considered with dealing with the star.
- Although its possible to use parentheses to group regular expressions, you should not be tempted to think that using `(se)*` is a good idea, because the regex compiler will interpret it as meaning "zero or more occurrences of se".
- If you apply this regex to the same string, you will encounter a total of 32 matches, because every character in the string would match the expression. (Remember? 0 or more occurrences!)

Regular Expressions In Python

- You'll find parentheses are often used in conjunction with the pipe operator to specify alternative regex specifications.
- For example, the regex `gr(u|a)b` with the string: "grab the grub and pull" would match both "grub" and "grab".
- Although regular expressions are quite powerful because of the original rules, inherent limitations make their use impractical.
- For example, there is no regular expression that can be used to specify the concept of "any character".
- As a result of the inherent limitations, the practical implementations of regex have grown to include a number of other rules, the most common of which are shown beginning on the next page.

Additional Syntax For Regex

- The special character "^" is used to identify the beginning of the string.
- The special character "$" is used to identify the end of the string.
- The special character "." is used to identify any character.
- Any nonnumeric character following the character "\" is interpreted literally (instead of being interpreted according to its regex meaning).

Additional Syntax For Regex

- Any regular expression followed by a "+" character is a regular expression composed of one or more instances of that regular expression.
- Any regular expression followed by a "?" character is a regular expression composed of either zero or one instance of that regular expression.
- Any regular expression followed by an expression of the type {min [, |, max]} is a regular expression composed of a variable number of instances of that regular expression. The min parameter indicates the minimum acceptable number of instances, whereas the max parameter, if present, indicates the maximum acceptable number of instances. If only the comma is present, no upper limit exists. If only min is defined, it indicates the only acceptable number of instances.
- Square brackets can be used to identify groups of characters acceptable for a given character position.

Some Basic Regex Usage

- It's sometimes useful to be able to recognize whether a portion of a regular expression should appear at the beginning or the end of a string.
- For example, suppose you're trying to determine whether a string represents a valid HTTP URL. The regex `http://` would match both http://www.cs.ucf.edu, which is valid and nhttp://www.cs.ucf.edu which is not valid, and could easily represent a typo on the user's part.
- Using the special character "^", you can indicate that the following regular expression should only be matched at the beginning of the string. Thus, `^http://` will match only the first of our two strings.

Some Basic Regex Usage

- The same concept – although in reverse – applies to the end-of-string marker "$, which indicates that the regular expression preceding it must end exactly at the end of the string.
- Thus, `com$` will match "amazon.com" but not "communication".
- Having a "wildcard" that can be used to match any character is extremely useful in a wide range of scenarios, particularly considering that the "." character is considered a regular expression in its own right, so that it can be combined with the Kleene Start and any of the other modifiers.

Some Basic Regex Usage

- Consider the regex: `.+@.+\..+`
- This regex can be used to indicate:
  - At least one instance of any character, followed by
  - The @ character, followed by
  - At least one instance of any character, followed by
  - The "." character, followed by
  - At least one instance of any character
- Can you guess what sort of string this regex might validate?

Does this look familiar? markl@cs.ucf.edu
It's a very rough form of an email address. Notice how the backslash character was used to force the regex compiler to interpret the next to last "." as a literal character, rather than as another instance of the "any character" regular expression.

Some Basic Regex Usage

- The regex on the previous page is a fairly crude way of checking the validity of an email address. After all, only letters of the alphabet, the underscore character, the minus character, and digits are allowed in the name, domain, and extension of an email.

- This is where the range denominators come into play. As mentioned previously (last paragraph of page 12), anything within non-escaped square brackets represents a set of alternatives for a particular character position. For example, the regex `[abc]` indicated either an "a", a "b", or a "c" character. However, representing something like "any character" by including every possible symbol in the square brackets would give rise to some ridiculously long regular expressions.

Some Basic Regex Usage

- Fortunately, range denominators make it possible to specify a "range" of characters by separating them with a dash.
- For example `[a-z]` means "any lowercase character.
- You can also specify more than one range and combine them with individual characters by placing them side-by-side.
- For example, our email validation regex could be satisfied by the expression `[A-Za-z0-9_]`.
- Using this new tool our full email validation expression becomes:

```
[A-Za-z0-9_]+@[A-Za-z0-9_]+\.[A-Za-z0-9_]+
```

Some Basic Regex Usage

- The range specifications that we have seen so far are all *inclusive* – that is, they tell the regex compiler which characters *can* be in the string. Sometimes, its more convenient to use *exclusive* specification, dictating that any character *except* the characters you specify are valid.
- This is done by prepending a caret character (^) to the character specifications inside the square bracket.
- For example, `[^A-Z]` means any character except any uppercase letter of the alphabet.

Some Basic Regex Usage

- Going back to our email example, its still not as good as it could be because we know for sure that a domain extension must have a minimum of two characters and a maximum of four.

- We can further modify our regex by using the minimum-maximum length specifier introduced on page 12.

```
[A-Za-z0-9_]+@[A-Za-z0-9_]+\.[A-Za-z0-9_]{2,4}
```

- Naturally, you might want to allow only email addresses that have a three-letter domain. This can be accomplished by omitting the comma and the `max` parameter from the length specifier, as in:

```
[A-Za-z0-9_]+@[A-Za-z0-9_]+\.[A-Za-z0-9_]{3}
```

Some Basic Regex Usage

- On the other hand, you might want to leave the maximum number of characters open in anticipation of the fact that longer domain extensions might be introduced in the future, so you could use the regex:

```
[A-Za-z0-9_]+@[A-Za-z0-9_]+\.[A-Za-z0-9_]{3,}
```

- Which indicates that the last regex in the expression should be repeated at least a minimum of three times, with no fixed upper limit.

A Practice Exercise

- See if you can create a regex that will validate a string representing a date in the format `mm/dd/yyyy`. In other words, 04/16/2012 would be matched but 4/16/12 would not.
- Step 1: form a basic regex. A regex such as .+ (one or more characters) is a bit too vague even as a starting point. So how about something like this?

```
[[0-9]]{2}/[[0-9]]{2}/[[0-9]]{4}
```

- This will work and validate 04/16/2012. However, it will also validate 99/99/2012 which is not a valid date, so we still need some refinement.

A Practice Exercise (continued)

- For the month component of our regex, the first digit must always be either a 0 or a 1, but the second digit can be any of 0 through 9.
- Similarly, for the day component of the regex, the first digit can only be 0, 1, 2, or 3.
- Our final regex now becomes:

```
[0-1][[0-9]]/[0-3][[0-9]]/[[0-9]]{4}
```

- This will work and validate 04/16/2012.

Perl-Compatible Regular Expressions (PCRE)

- Perl-Compatible Regular Expressions (PCRE) are much more powerful than their basic regex (or POSIX) counterparts. This of course makes them more complex and difficult to use as well, but well worth the effort for Python programmers/scripters.
- PCRE adds its own character classes to extend regular expressions.
- There are nine of these character classes in PCRE and are shown in the table on the next page.

Perl-Compatible Regular Expressions (PCRE)

Character class	Description
\w	Represents a "word" character and is equivalent to the expression `[A-Za-z0-9]`
\W	Represents the opposite of `\w` and is equivalent to the expression `[^A-Za-z0-9]`
\s	Represents a whitespace character
\S	Represents a non-whitespace character
\d	Represents a digit and is equivalent to the expression `[0-9]`
\D	Represents a non-digit (the opposite of `\w`) and is equivalent to the expression `[^0-9]`
\n	Represents a new line character
\r	Represents a return character
\t	Represents a tab character

PCRE character classes

Perl-Compatible Regular Expressions (PCRE)

- Using PCRE formatted regex allows for significantly more concise regex than is possible for the POSIX formatted regex.
- Consider, for example, the email address validation expression we developed in POSIX earlier:

```
[[0-9A-Za-z]_]+@[[0-9A-Za-z]_]+\.[[0-9A-Za-z]_]{2,4}
```

- Using the new character classes of PCRE this expression becomes:

```
/\w+@\w+\.\w{2,4}/
```

Note in the example script on page 26 I added another \.\w{2,4} term so that I could easily pickup the sub-domain used in my email address.

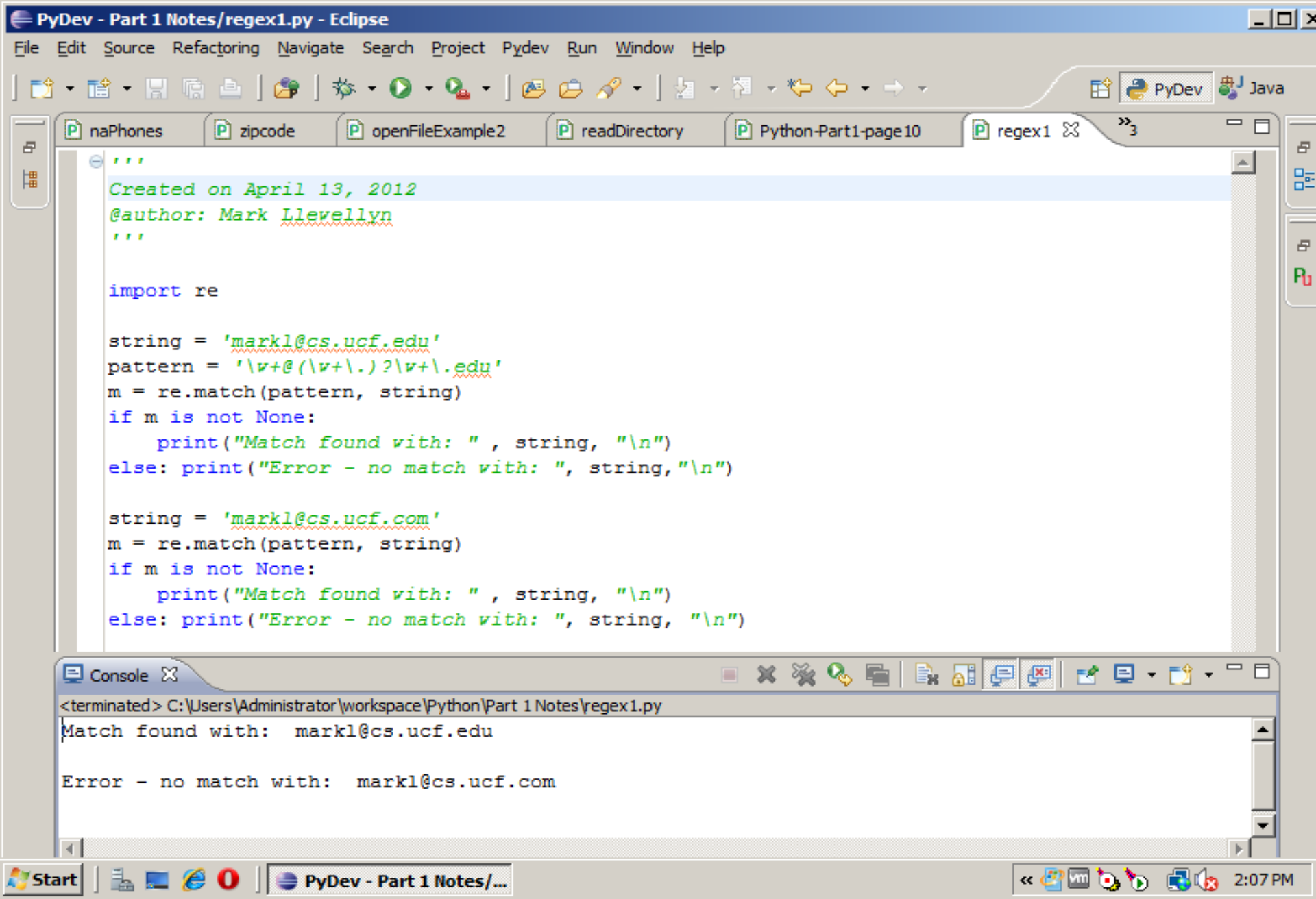
Notice that the regex string now begins and ends with forward slashes. PCRE requires that the actual regular expression be delimited by two characters. By convention, two forward slashes are used, although any character other than the backslash that is not alphanumeric would work just as well.

Perl-Compatible Regular Expressions (PCRE)

- Regular expressions are supported in Python through the `re` module.
- There are a lot of functions in this module, but the primary function for regular expressions is `match()`. This function has the following syntax:

```
match(pattern, string, flags=0)
```

- The `match()` function attempts to match the regular expression `pattern` to `string` with optional `flags`; returns a `match` object on success, `None` on failure.
- An example illustrating this function is shown on the next page.


PyDev - Part 1 Notes/regex1.py - Eclipse
File Edit Source Refactoring Navigate Search Project Pydev Run Window Help
PyDev
Java
naPhones
zipcode
openFileExample2
readDirectory
Python-Part1-page10
regex1
'''
Created on April 13, 2012
@author: Mark Llewellyn
'''
import re
string = 'markl@cs.ucf.edu'
pattern = '\w+@(\w+\.)?\w+\.edu'
m = re.match(pattern, string)
if m is not None:
print("Match found with: " , string, "\n")
else: print("Error - no match with: ", string,"\n")
string = 'markl@cs.ucf.com'
m = re.match(pattern, string)
if m is not None:
print("Match found with: " , string, "\n")
else: print("Error - no match with: ", string, "\n")
Console
<terminated> C:\Users\Administrator\workspace\Python\Part 1 Notes\regex1.py
Match found with: markl@cs.ucf.edu
Error - no match with: markl@cs.ucf.com
Start
PyDev - Part 1 Notes/...
2:07 PM

Perl-Compatible Regular Expressions (PCRE)

- Some of the more common Python functions that are part of the `re` module are shown below:

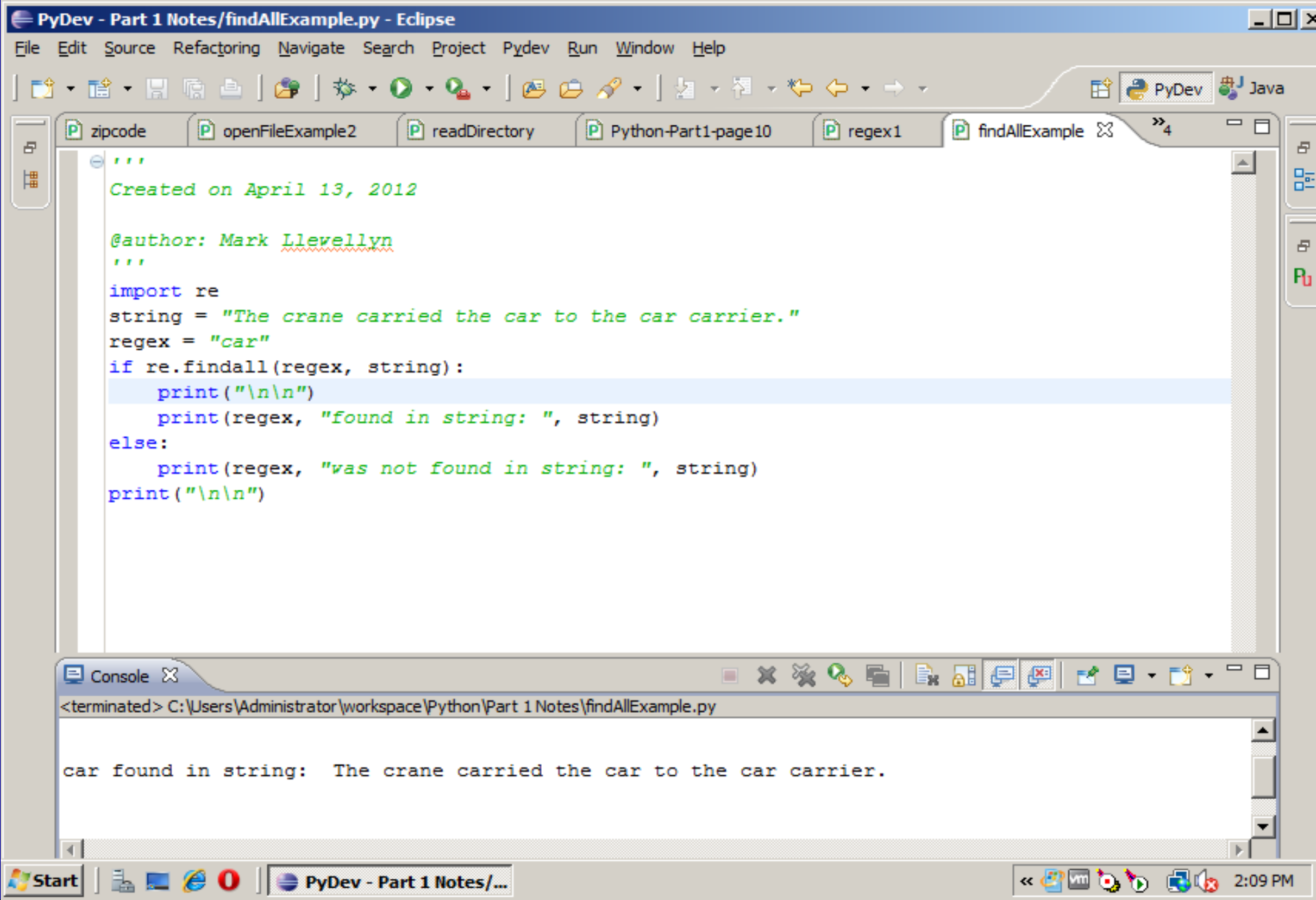
`search(pattern, string, flags=0)` – searches for the first occurrence of `pattern` within `string`. Returns match object on success, None on failure.

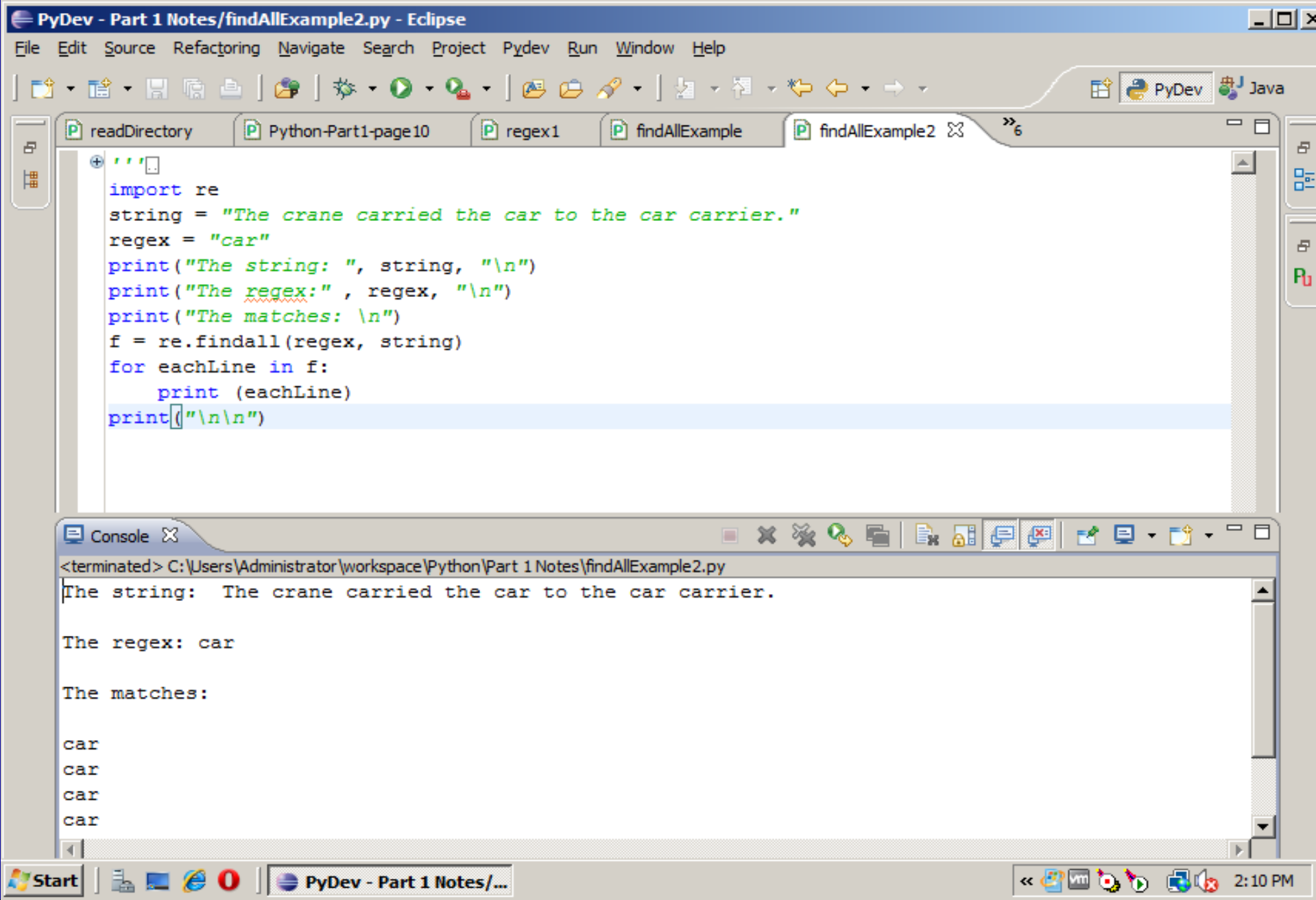
`findall(pattern, string [, flags])` – looks for all (non-overlapping) occurrences of `pattern` in `string` and returns a list of matches, possibly empty if none are found.

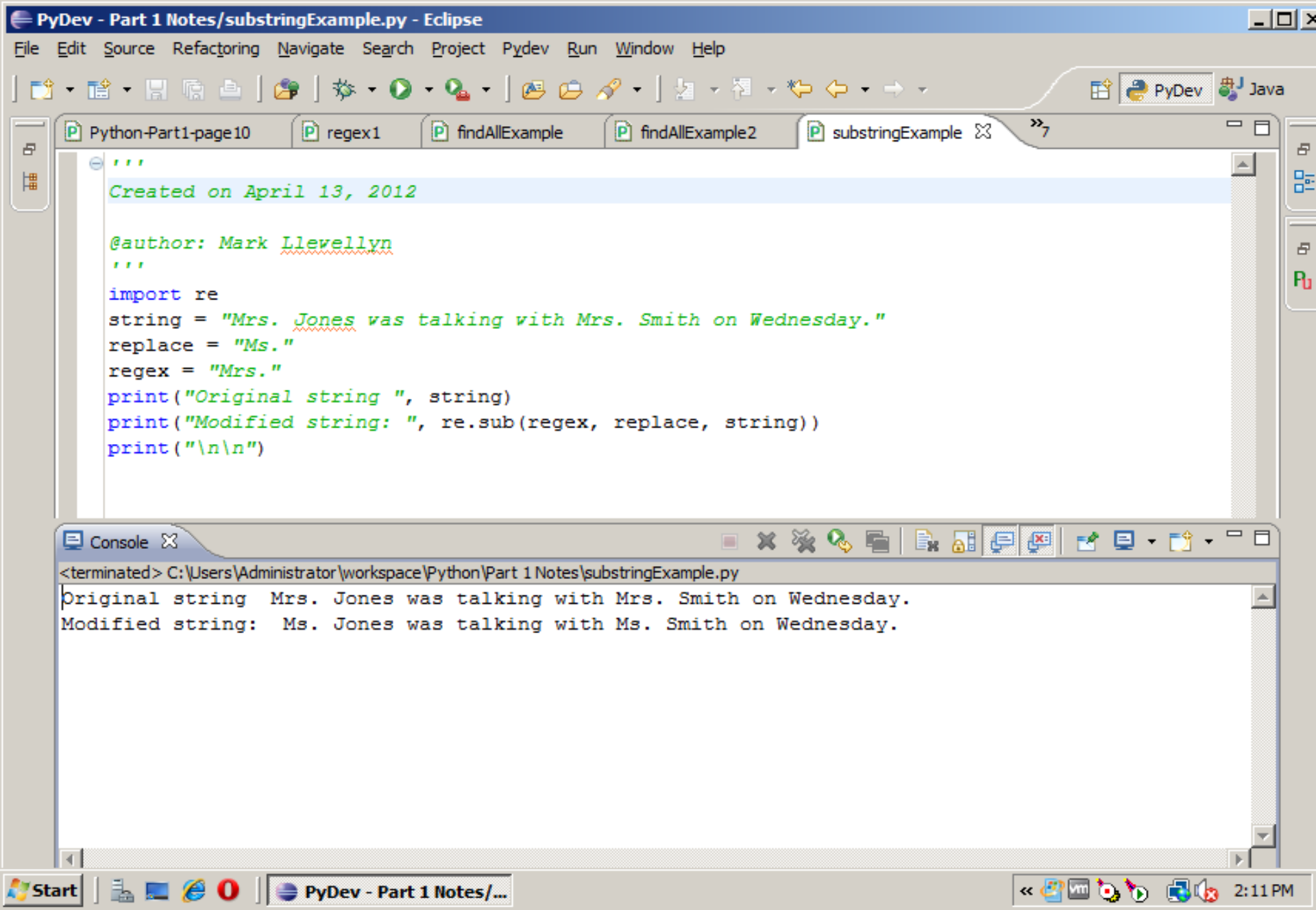
`finditer(pattern, string [, flags])` – same as `findall()` except returns an iterator instead of a list; for each match, the iterator returns a match object.

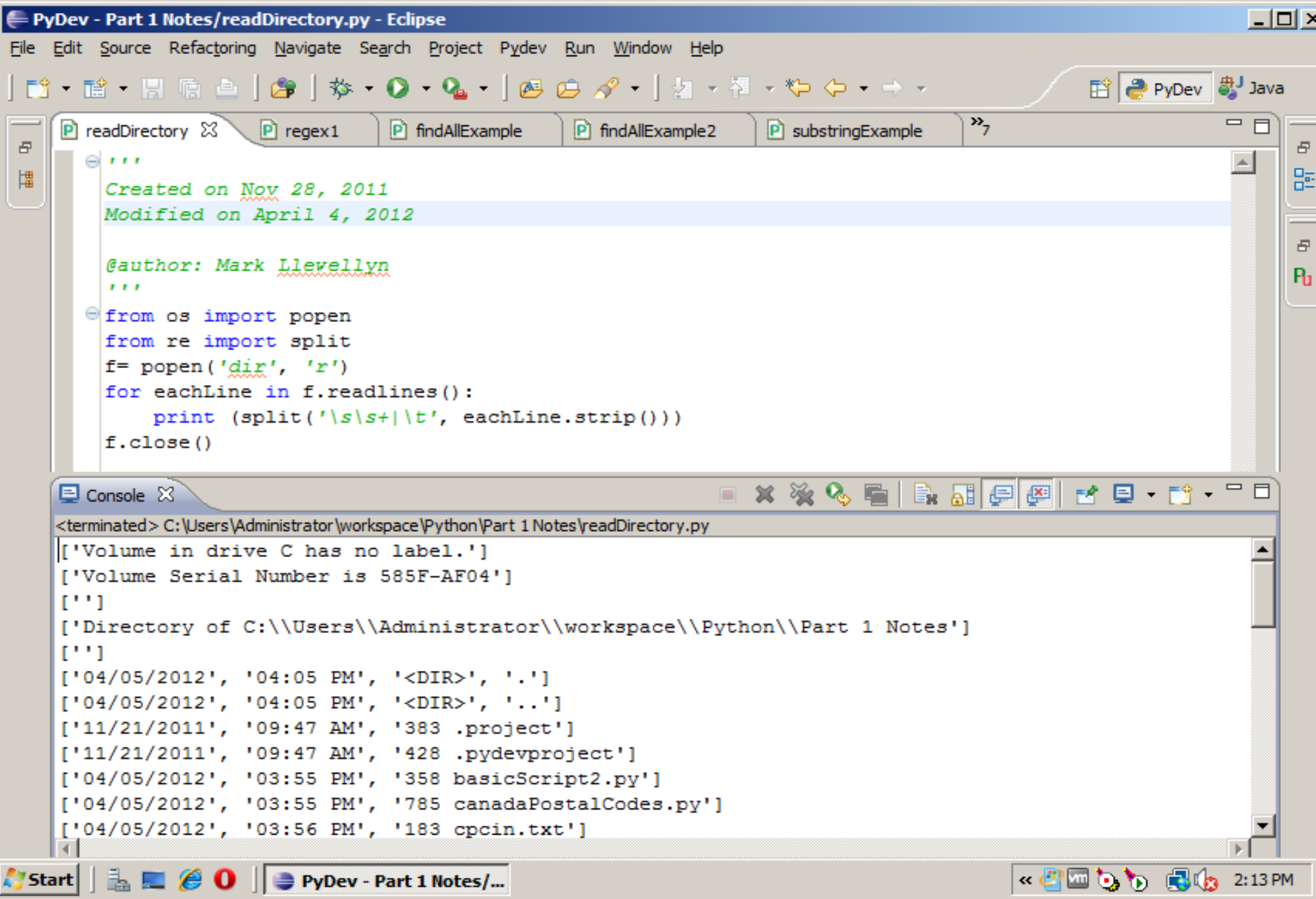
`split(pattern, string, max=0)` – splits `string` into a list according to the `pattern` delimiter and returns a list of successful matches, splitting at most `max` times (split all is the default).

`sub(pattern, replace, string, max=0)` – replaces all occurrences of `pattern` in `string` with `replace`, substituting all occurrences unless `max` is provided.


PyDev - Part 1 Notes/findAllExample.py - Eclipse
File Edit Source Refactoring Navigate Search Project Pydev Run Window Help
PyDev
Java
zipcode
openFileExample2
readDirectory
Python-Part1-page10
regex1
findAllExample
'''
Created on April 13, 2012
@author: Mark Llewellyn
'''
import re
string = "The crane carried the car to the car carrier."
regex = "car"
if re.findall(regex, string):
print("\n\n")
print(regex, "found in string: ", string)
else:
print(regex, "was not found in string: ", string)
print("\n\n")
Console
<terminated> C:\Users\Administrator\workspace\Python\Part 1 Notes\findAllExample.py
car found in string: The crane carried the car to the car carrier.
Start
PyDev - Part 1 Notes/...
2:09 PM


PyDev - Part 1 Notes/findAllExample2.py - Eclipse
File Edit Source Refactoring Navigate Search Project Pydev Run Window Help
PyDev
Java
readDirectory
Python-Part1-page10
regex1
findAllExample
findAllExample2
'''
import re
string = "The crane carried the car to the car carrier."
regex = "car"
print("The string: ", string, "\n")
print("The regex:" , regex, "\n")
print("The matches: \n")
f = re.findall(regex, string)
for eachLine in f:
print (eachLine)
print("\n\n")
Console
<terminated> C:\Users\Administrator\workspace\Python\Part 1 Notes\findAllExample2.py
The string: The crane carried the car to the car carrier.
The regex: car
The matches:
car
car
car
car
Start
PyDev - Part 1 Notes/...
2:10 PM


PyDev - Part 1 Notes/substringExample.py - Eclipse
File Edit Source Refactoring Navigate Search Project Pydev Run Window Help
PyDev
Java
Python-Part1-page10
regex1
findAllExample
findAllExample2
substringExample
'''
Created on April 13, 2012
@author: Mark Llewellyn
'''
import re
string = "Mrs. Jones was talking with Mrs. Smith on Wednesday."
replace = "Ms."
regex = "Mrs."
print("Original string ", string)
print("Modified string: ", re.sub(regex, replace, string))
print("\n\n")
Console
<terminated> C:\Users\Administrator\workspace\Python\Part 1 Notes\substringExample.py
Original string Mrs. Jones was talking with Mrs. Smith on Wednesday.
Modified string: Ms. Jones was talking with Ms. Smith on Wednesday.
Start
PyDev - Part 1 Notes/...
2:11 PM


PyDev - Part 1 Notes/readDirectory.py - Eclipse
File Edit Source Refactoring Navigate Search Project Pydev Run Window Help
PyDev
Java
readDirectory
regex1
findAllExample
findAllExample2
substringExample
'''
Created on Nov 28, 2011
Modified on April 4, 2012
@author: Mark Llewellyn
'''
from os import popen
from re import split
f= popen('dir', 'r')
for eachLine in f.readlines():
print (split('\s\s+|\t', eachLine.strip()))
f.close()
Console
<terminated> C:\Users\Administrator\workspace\Python\Part 1 Notes\readDirectory.py
['Volume in drive C has no label.']
['Volume Serial Number is 585F-AF04']
['']
['Directory of C:\\Users\\Administrator\\workspace\\Python\\Part 1 Notes']
['']
['04/05/2012', '04:05 PM', '<DIR>', '.']
['04/05/2012', '04:05 PM', '<DIR>', '..']
['11/21/2011', '09:47 AM', '383 .project']
['11/21/2011', '09:47 AM', '428 .pydevproject']
['04/05/2012', '03:55 PM', '358 basicScript2.py']
['04/05/2012', '03:55 PM', '785 canadaPostalCodes.py']
['04/05/2012', '03:56 PM', '183 cpcin.txt']
Start
PyDev - Part 1 Notes/...
2:13 PM

Practice Problems

- See if you can generate regular expressions to recognize the following strings:

  1. To recognize the any of the words: “bat”, “bit”, “but”, “hat”, “hit”, or “hut”

  2. Any pair of words separated by a space or a colon. Example: “alpha:beta”

  3. Any floating point number represented by any number of digits followed optionally by a single decimal point and zero or more digits. Examples: 45.6, 0.004, 75

  4. Any day of the week, e.g. “Monday”

- The solutions appear on the next two pages…try these problems yourself **BEFORE** you look at the solutions.

Practice Problems - **Solutions**

1. To recognize the any of the words: “bat”, “bit”, “but”, “hat”, “hit”, or “hut”

   `(bat | bit | but | hat | hit | hut)` //not very elegant!

   `[bh][aiu]t`

   `(b|h)(a|i|u)t`

2. Any pair of words separated by a space or a colon. Example: “alpha:beta”

   `\w+(:|\s)\w`

Practice Problems - **Solutions**

3. Any floating point number represented by any number of digits followed optionally by a single decimal point and zero or more digits. Examples: 45.6, 0.004, 75

```
[0-9]+(\.[0-9]*)?
```

```
\d+(\.\d*)?
```

4. Any day of the week

```
(Mon | Tues | Wed | Thurs | Fri | Satur | Sun)day
```