Python Advanced Programming

Unit 1: Tokenization

CHAPTER 3: REGULAR EXPRESSION (PYTHON)

DR. ERIC CHOU

IEEE SENIOR MEMBER

Introduction to Python Regular Expression

LECTURE 1

Introduction

- **Regular Expressions** are patterns that we can specify and use to search and replace text in strings (and files, which are just a sequence of strings).
- Python includes a module to perform various operations on regular expressions. In this Chapter, we will cover
  1. **the form of regular expressions**,
  2. **what functions/methods can take regular expressions as arguments**, and
  3. **how to use the results of matching regular expressions against text** (including the concept of match groups).

Topics

- In the first part we will discuss the components of regular expression patterns.
- Each component individually, and the ways to combine them into more complicated regular expressions (just as we studied the syntax of a few simple control structures
- in Python, which can be combined into more complicated control structures).

Topics

- In the second part, we will examine functions and methods that take regular expressions as arguments and produce results.
- Typically they match a regular expression pattern against some text string, and return information about whether or not the match succeeded, and what parts of the pattern matched which parts of the text.

Topics

- Python's module for doing these operations is named **re**. There is a special tester module that accompanies these lectures, which you can download and run to experiment with regular expressions and learn how they match text strings.
- We will also look at an online resource for learning and testing regular expressions.
- For more complete information about regular expressions, see Section 6.2 of the Python Standard Library.

Regular Expression Rules

LECTURE 1

General Rule of Matching:

- **General Rule of Matching:** Regular expressions match the most number of characters possible (called a **greedy algorithm**; there are patterns that match the fewest number of characters possible; we will mention, but not discuss nor use those patterns).

Regular Expressions

Metacharacters

.	Matches any single character (except newline: \n)
[]	Matches one character specified inside []; e.g., [aeiou]
[^]	Matches one character NOT specified inside [] after ^; e.g., [^aeiouy]
-	Matches one character in range inside []: e.g., [0-9] matches any digit

Anchors (these don't match characters)

^	matches beginning of line (when not used in [])
$	matches end of line

Regular Expressions

Patterns: R, Ra, Rb are regular expression patterns

RaRb Matches a sequence of Ra followed by Rb

Ra|Rb Matches either alternative Ra or Rb

R? Matches regular expression R 0/1 time: R is optional

R* Matches regular expression R 0 or more times

R+ Matches regular expression R 1 or more times

R{m} Matches regular expression R exactly m times: e.g., R{5} = RRRRR

R{m, n} Matches regular expression R at least m and at most n times:

R{3,5} = RRR|RRRR|RRRRR = RRRR?R?

R??,R*?,R+?,R{m,n}? The postfix ? means match as few characters possible (not the most: so not greedy).

Parentheses/Parenthesized Patterns

- Parentheses are used for grouping, but can also remember subpatterns (this is also called a "**Capturing Group**").
- By placing subpattern **R** in parentheses, the text matching **R** will be remembered (either by its number, starting at 1, or its name, if named) in a group, for use later in the pattern or when extracting information from the matched text.

Parentheses/Parenthesized Patterns

(R) Matches R and delimits a group (1...) (remembers/captures matched text)

(?P<name>R) Matches R and remembers/captures matched text in a group using name for the group (it is still numbered as well); see **(?P=name)** and groupdict method below for use of "name".

(?:R) Matches R but does not remember/capture matched text in a group So, there () are used only for grouping, not capturing groups; **?:** is useful when you want the minimum number (no redundant groups)

Parentheses/Parenthesized Patterns

(?P=name) Matches remembered text with name (for backreferencing which text)

(?=R) Matches **R**, doesn't remember matched text/consume text matched. For example (?=abc)(.*) matches abcxyz with group 1 'abcxyz'; it doesn't match abxy because this text doesn't start with abc

(?!R) Matches anything but **R** and does not consume input needed for match (hint: **!=** means "not equal", **?!R** means "not matching **R**)

Context

- matches itself if not in [], and if not between two characters

- Special characters are treated as themselves in []: e.g, [.] matches literal .
- Generally, if interpreting a character makes no sense one way, try to find another way to interpret it that fits the context

Escape Characters with Special Meanings

****	Used before .|[]-?*+{}()^$\ (and others) to specify a special character
\#	Backreferencing group # (numbered from 1, 2, ...): see (R) above
\t	tab
\n	newline
\r	carriage return
\f	form-feed
\v	vertical tab

Escape Characters with Special Meanings

\d	[0-9]	Digit
\D	[^0-9]	non-Digit
\s	[\t\n\r\f\v]	White space
\S	[^ \t\n\r\f\v]	non-White space
\w	[a-zA-Z0-9_]	alphabetic (or underscore): Word character (id letters)
\W	[^a-zA-Z0-9_]	non-alphabetic: non-Word character (non-id letters)

Interesting Equivalences

a+	**== aa***	
a(b|c|d)e	**== a[bcd]e**	only if b, c, and d are single characters
R{0,1}	**== R?**	

Hints on Using |

(a low-precedence operator for Regular Expression)

- In Python, we know that writing a*b+c*d performs * before +: we say * has higher precedence than +, so it is performed earlier.
- We could be explicit and write this as (a*b)+(c*d). Think of REs as having sequence as an operation (it is implicit, with no operator).
- The seqeuence precendence is lower than the precedence of postfix operators (like ?, *, and +): e.g., ab* has the same meaning as a(b*).
- **All have higher precdence than |.** So writing ab|cd is the equivalent of writing (ab)|(cd), which matches either ab or cd only.

Hints on Using |

(a low-precedence operator for Regular Expression)

- Now, given this understanding, look what **^a|b$** means. By above, it means the same as **(^a)|(b$)**. Type **^a|b$** into the online tool and read its Explanation).
- Note that the **^** anchor applies only to a, and the **$** anchor applies only to **b** (see its parenthesized equivalent). So **^a|b$** (which is equivalent to **(^a)|(b$)** will match (using the online tool)
  - **any text starting with an a: a** or **a**a or **a**aab or **a**bcda ($ is not part of it)
  - **any text ending with a b: b** or c**b** or ccb**b** or abcd**b** (^ is not part of it)

Hints on Using |

(a low-precedence operator for Regular Expression)

- To avoid confusion, I strongly recommend always writing all the alternatives in a regular expression as a group: **^(a|b)$** to ensure that the **|** applies only to the alternatives inside the **()**s.
- This regular expression is a sequence of 3: the **^** anchor, a choice of a or b, and a **$** anchor.

Cheat Sheet

LECTURE 1

Anchors

^	Start of line
$	End of line

Character Classes

\s	White space character
\S	Non-white space character
\d	Digit character
\D	Non-digit character
\w	Word
\W	Non-word (e.g. punctuation, spaces)

Metacharacters (must be escaped)

^	[	]
$	(	)
.	{	}
*	+	?
\	|	-

GA Filter group accessors

$Ax	Access group x in field A (e.g. $A1)
$Bx	Access group x in field B (e.g. $B1)

Quantifiers

*	Zero or more (greedy)
*?	Zero or more (lazy)
+	One or more (greedy)
+?	One or more (lazy)
?	Zero or one (greedy)
??	Zero or one (lazy)
{X}	Exactly X (e.g. 3)
{X,}	X or more, (e.g. 3)
{X, Y}	Between X and Y (e.g. 3 and 5) (lazy)

Ranges and Groups

.	Any character
(a|b)	a or b (case sensitive)
(...)	Group, e.g. (keyword)
(?:...)	Passive group, e.g. (?:keyword)
[abc]	Range (a or b or c)
[^abc]	Negative range (not a or b or c)
[A-Z]	Uppercase letter between A and Z
[a-z]	Lowercase letter between a and z
[0-7]	Digit between 0 and 7

Sample Patterns

^/directory/(.*)
Any page URLs starting with /directory/

(brand\s*?term)
Brand term with or without whitespace between words

^brand\s+[^cf]
Key phrases beginning with 'brand' and the second word not starting with c or f

\.aspx$
URLs ending in '.aspx'

ORDER\-\d{6}
"ORDER-" followed by a six digit ID

(?:\?|&)utm=([^&$]+)
Value of 'utm' querystring parameter

eC Learning Channel

Example by Java/Python (Basic Regex)

LECTURE 1

Regular Expression Syntax

[or-set] [^not-set], Range: [a-z], Union: [set1[set2]] Intersection: [set1&&set2]

Regular Expression	Matches	Example
x	a specified character x	Java matches Java
.	any single character	Java matches J..a
(ab|cd)	a, b, or c	ten matches t(en|im]
[abc]	a, b, or c	Java matches Ja[uvwx]a
[^abc]	any character except a, b, or c	Java matches Ja[^ars]a
[a-z]	a through z	Java matches [A-M]av[a-d]
[^a-z]	any character except a through z	Java matches Jav[^b-d]
[a-e[m-p]]	a through e or m through p	Java matches [A-G[I-M]]av[a-d]
[a-e&&[c-p]]	intersection of a-e with c-p	Java matches [A-P&&[I-M]]av[a-d]

Regular Expression Syntax

\one-wildcard-type-letter (in string \\d, first \ is escape)

Regular Expression	Matches	Example
\d	a digit, same as [1-9]	Java2 matches "Java[\\d]"
\D	a non-digit	$Java matches "[\\D][\\D]ava"
\w	a word character	Java matches "[\\w]ava"
\W	a non-word character	$Java matches "[\\W][\\w]ava"
\s	a whitespace character	"Java 2" matches "Java\\s2"
\S	a non-whitespace char	Java matches "[\\S]ava"
Quantifiers		
*p**	zero or more occurrences of pattern *p*	Java matches "[\\w]*"
p+	one or more occurrences of pattern *p*	Java matches "[\\w]+"
p?	zero or one occurrence of pattern *p*	Java matches "[\\w]?Java" Java matches "[\\w]?ava"
p{n}	exactly n occurrences of pattern *p*	Java matches "[\\w]{4}"
p{n,}	at least n occurrences of pattern *p*	Java matches "[\\w]{3,}"
p{n,m}	between n and m occurrences (inclusive)	Java matches "[\\w]{1,9}"

Python re.match()

Demo Program: regex_match.py

re.match():

re.match(pattern, string[, flags])

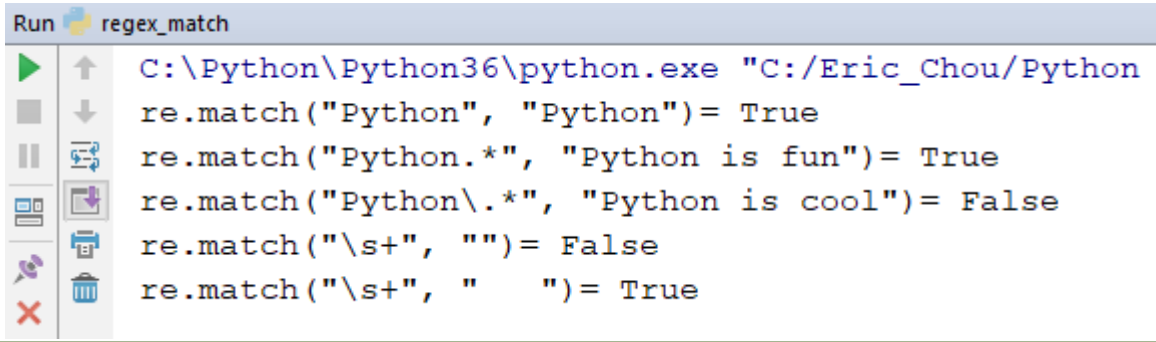
Example:

match-object = re.match(pattern, string_to_be_matched)

Return Value:

Return None if the string does not match the pattern; note that this is different from a zero-length match.

```python
import re                                                    # regex_match.py
# matches Python Strings
matchObj= re.match("Python", "Python")
if matchObj == None: matching = False
else: matching = True
print("re.match(\"Python\", \"Python\")=", matching)
# matches zero or more characters
matchObj= re.match("Python.*", "Python is fun")
if matchObj == None: matching = False
else: matching = True
print("re.match(\"Python.*\", \"Python is fun\")=", matching)
# not matching \.
matchObj= re.match("Python\.+", "Python is cool")
if matchObj == None: matching = False
else: matching = True
print("re.match(\"Python\\.*\", \"Python is cool\")=", matching)
# not matching one or more space
matchObj= re.match("\s+", "")
if matchObj == None: matching = False
else: matching = True
print("re.match(\"\\s+\", \"\")=", matching)
# matching one or more space
matchObj= re.match("\s+", "    ")
if matchObj == None: matching = False
else: matching = True
print("re.match(\"\\s+\", \"    \")=", matching)
```



Regular Expression Designs (Part 1)

LECTURE 1

Design Regular Expression for the Patterns

- Write the smallest pattern that matches the required characters.
- Check your patterns with the Regular Expression Tester (see the Sample Programs **link**) to ensure they match correct exemplars and don't match incorrect ones.
- Note that for a match, group #0 should include all the required characters.

Problem 1:

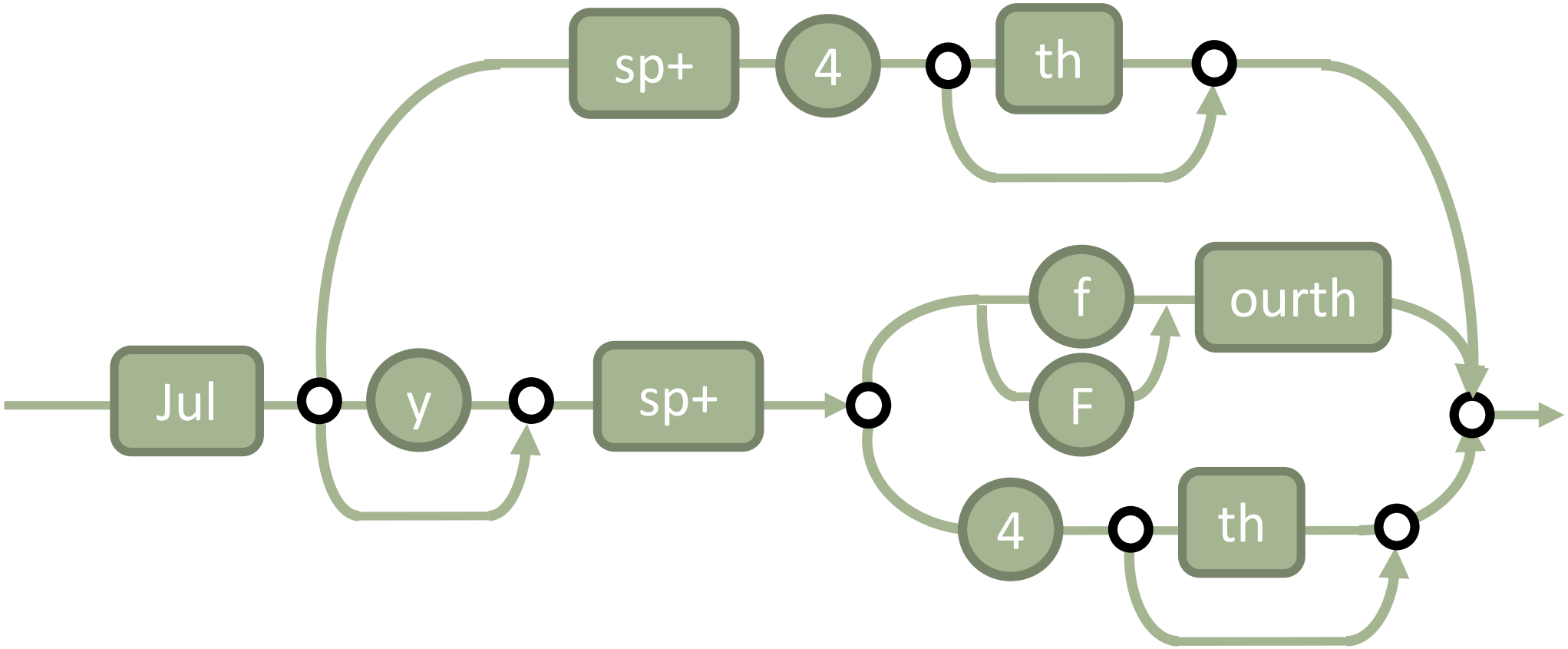
1. Write a regular expression pattern that matches the strings **Jul 4, July 4, Jul 4th, July 4th, July fourth, and July Fourth**.

   Hint: my re pattern was 24 characters.

Problem 1 Solution: july_fourth.py

```
# july_fourth.py
# problme 1:
# regular expression for July Fourth
import re
# Sample strings.
list = ["Jul 4", "Jul 4th", "July 4", "July 4th", "Jul Fourth",
        "July Fourth", "jul 4", "July fourth", "j 4", "july Fourth"]
# Loop.
for element in list:
     # Match if two words starting with letter d.
                    #12345678901234567890123
     m = re.match("Jul(\s4(th)?|y\s(4(th)?|(f|F)ourth))", element)
     # See if success.
     if m:
          print(m.group(0))
```

Syntax Diagram



Problem 2:

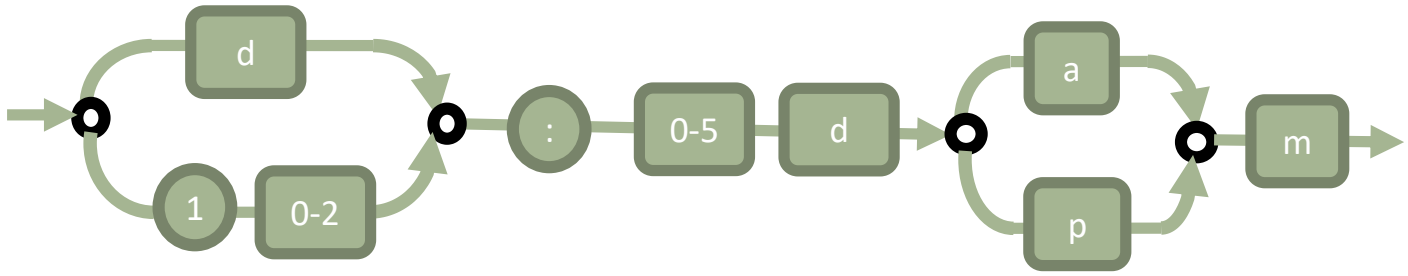
2. Write a regular expression pattern that matches strings representing **times on a 12 hour clock**.

An example time is 5:09am or 11:23pm. Allow only times that are legal (not 1:73pm nor 13:02pm)

Hint: my re pattern was 32 characters.

Problem 2 Solution: clock.py

```
# clock.py
# problem 2:
# regular expression for clock
import re
# Sample strings.
list = ["1:30pm", "23:18am", "32:00kk", "7:20pm", "10:32am"]
# Loop.
for element in list:
    # Match if two words starting with letter d.
            #12345678901234567890123456789012
    m = re.match("([1-9]|1[0-2]):[0-5][0-9](a|p)m", element)
    # See if success.
    if m:
        print(m.group(0))
```




```
Run clock
C:\Python\Python36\python.exe
1:30pm
7:20pm
10:32am
```

Problem 3:

3. Write a regular expression pattern that matches strings representing **phone numbers** of the following form.

- **Normal:** a three digit exchange, followed by a dash, followed by a four digit number: e.g., 555-1212
- **Long Distance:** a 1, followed by a dash, followed by a three digit area code enclosed in parentheses, followed by a three digit exchange, followed by a dash, followed by a four digit number: e.g.,

1-(800)555-1212

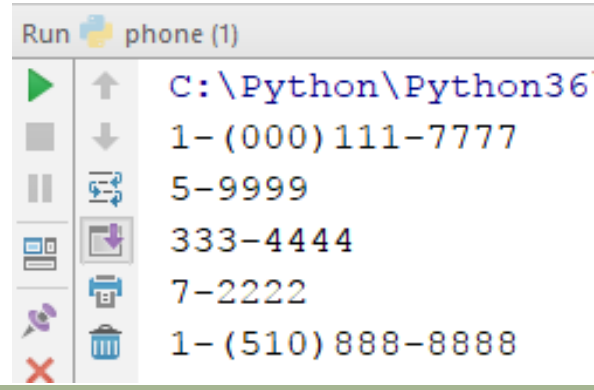
Problem 3:

- **Interoffice:** a single digit followed by a dash followed by a four digit number: e.g., 8-2404.

  Hint: my re pattern was 30 characters; note that you must use \(and \) to match parentheses.

Problem 3 Solution: phonep3.py

```
# phonep3.py
# problme 3:
# regular expression for phone number
import re
# Sample strings.
list = ["1-(000)111-7777", "0000","5-9999", "333-4444",
        "9999999", "7-2222", "1-(510)888-8888"]
# Loop.
for element in list:
     # Match if two words starting with letter d.
                    #123456789012345678901234567890123
     m = re.match("((1-\(\d{3}\))?\d\d)?\d-\d{4}", element)
     # See if success.
     if m:
          print(m.group(0))
```



Regex for Phone Call Formats

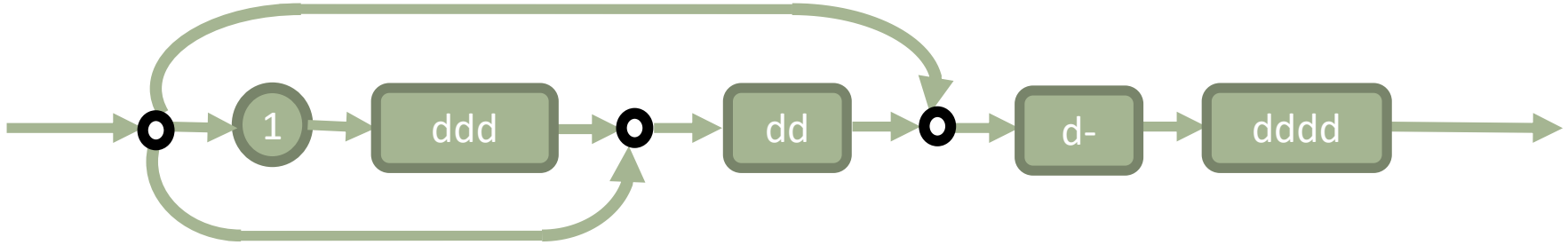
5-1212 Interoffcie = [0-9]-[0-9]{4}

555-1212 Normal = [0-9]{2}<Interoffice>

1-(800)555-1212 International = 1-\([0-9]{3}\)<Normal>

Three-in-one format:

(((1-\(<Area>\)?)[0-9]{2})?<Interoffice>

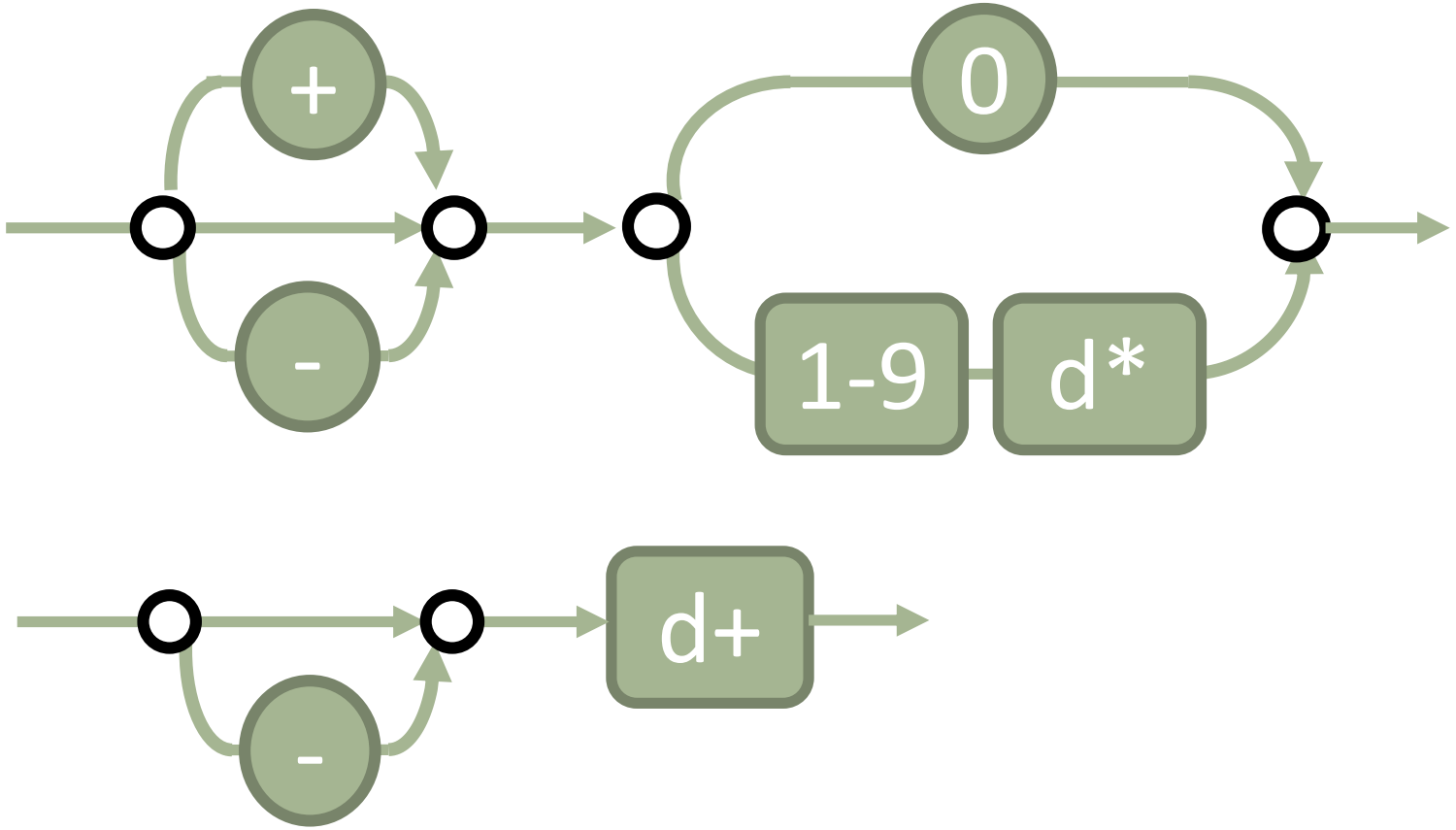


Problem 4:

4. Write a regular expression pattern that matches strings representing **simple integers**: optional + or - signs followed by one or more digits.

Hint: my re pattern was 7 characters.

Regex for Integer



Problem 4 Solution: integer.py

```
# integer.py
# problme 4:
# regular expression normal integer
import re
# Sample strings.
list = ["00", "000", "33","-102", "3.55", "0", "-32", "+255", "4300"]
# Loop.
for element in list:
      # Match if two words starting with letter d.
                  #12345678901234567890
      m = re.match("^(\+|-)?(0|([1-9]\d*))$", element)
      # See if success.
      if m:
            print(m.group(0))
```

Problem 4 Solution: integer_short.py

```
# integer_short.py
# problme 4:
# regular expression for short integer format
import re
# Sample strings.
list = ["00", "000", "33","-102", "3.55", "0", "-32", "+255",
"4300"]
# Loop.
for element in list:
      # Match if two words starting with letter d.
                  #12345678901234567890
      m = re.match("^-?\d+$", element)
      # See if success.
      if m:
            print(m.group(0))
```

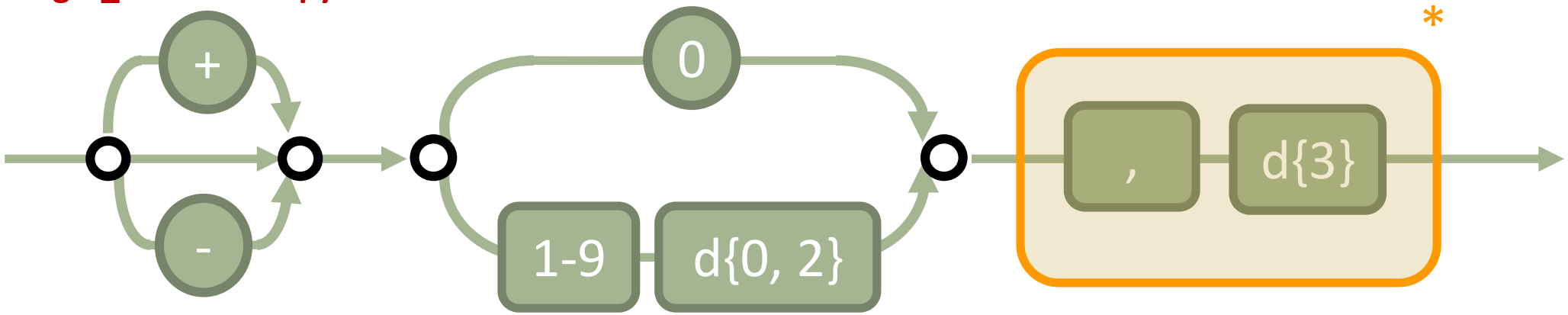
Problem 5:

5. Write a regular expression pattern that matches strings representing **normalized integers** (each number is either an unsigned 0 or is unsigned or signed and starts with a non-0 digit) with commas in only the correct positions

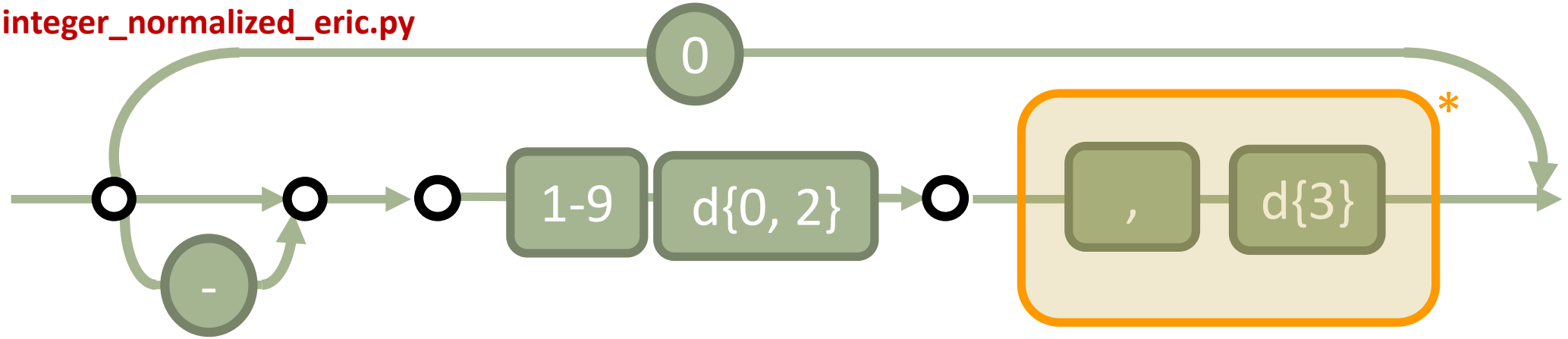
Hint: my re pattern was 30 characters.

Regex for Normalized Integer

integer_normalized.py



integer_normalized_eric.py



Demo Program: integer_normalized.py

```
# integer_normalized.py
# problme 5:
# regular expression normalized integer
import re
# Sample strings.
list = ["00", "1,000", "33","-102", "3.55", "0", "-32", "+255",
        "4,300", "1,000,000", "22,999,444"]
# Loop.
for element in list:
      # Match if two words starting with letter d.
                  #12345678901234567890
      m = re.match("^(\+|-)?(0|([1-9]\d*))(\,\d{3})*$", element)
      # See if success.
      if m:
            print(m.group(0))
```

Problem 6:

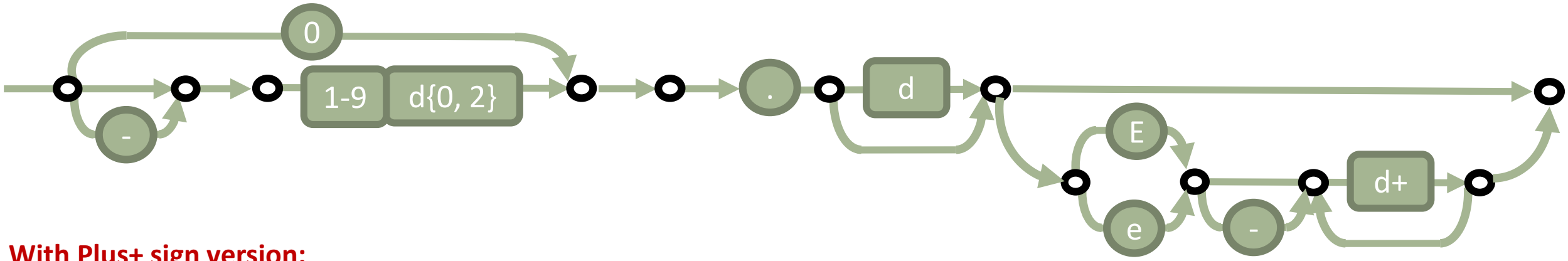
6. Write a regular expression pattern that matches strings representing **float values**.

They are unsigned or signed (but not normalized: see 5) and any number of digits before or after a decimal point (but there must be at least one digit either before or after a decimal point: e.g., just . is not allowed) followed by an optional e or E followed by an unsigned or signed integer (again not normalized).

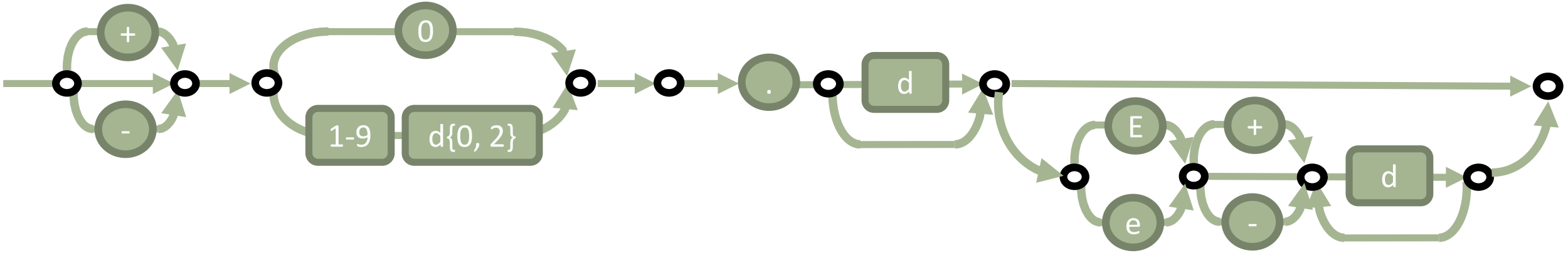
Hint: my re pattern was 36 characters.

Regex for Floating Point Number

Without Plus+ sign version:



With Plus+ sign version:



Problem 6 Solution: floatingpoint.py

```
# floatingpoint.py
# problme 6:
# regular expression floating point number
import re
# Sample strings.
list = ["0", "0.00", "100.0", "0.33","-102.5", "3.55", "0.0", "33",
        "-32.7423897423", "+2.55", "43.0e33", "3.22E+7843"]
# Loop.
for element in list:
     # Match if two words starting with letter d.
                    #12345678901234567890123456789012345678901234
     m = re.match("^(\+|-)?(0|([1-9]\d*))\.\d+((E|e)(\+|-)?\d+)?$", element)
     # See if success.
     if m:
        print(m.group(0))
```

Problem 7:

7. Write a regular expression pattern that matches strings representing **trains**.

A single letter stands for each kind of car in a train: Engine, Caboose, Boxcar, Passenger car, and Dining car. There are four rules specifying how to form trains.

1. One or more Engines appear at the front; one Caboose at the end.
2. Boxcars always come in pairs: BB, BBBB, etc.
3. There cannot be more than four Passenger cars in a series.
4. One dining car must follow each series of passenger cars.

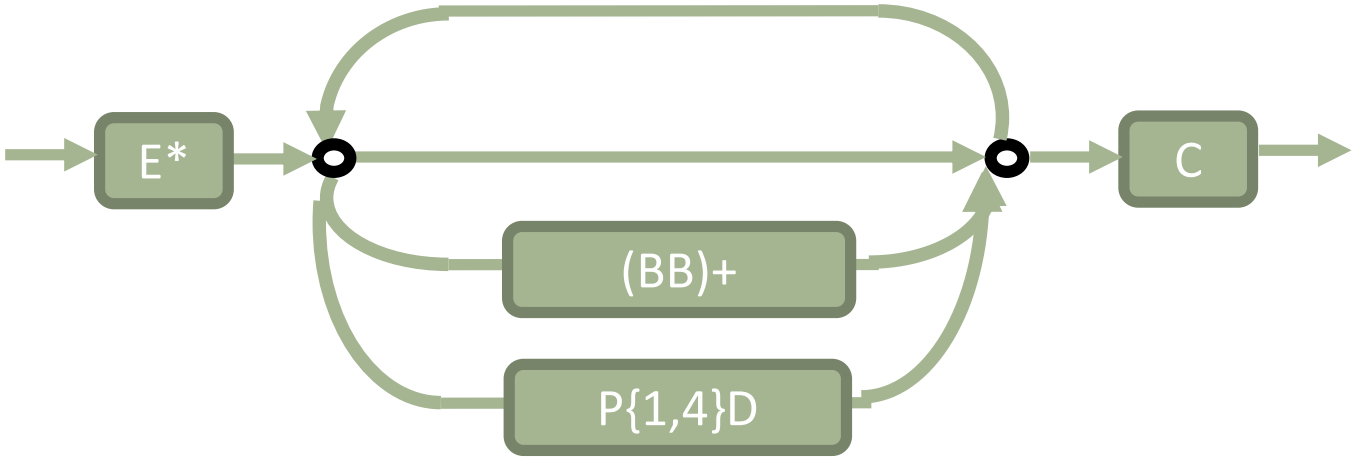
These cars cannot appear anywhere other than these locations. Here are some legal and illegal exemplars.

Problem 7:

EC	**Legal:** the smallest train
EEEPPDBBPDBBBBC	**Legal:** simple train showing all the cars
EEBB	**Illegal:** no caboose (everything else OK)
EBBBC	**Illegal:** three boxcars in a row
EEPPPPPDBBC	**Illegal:** more than four passenger cars in a row
EEPPBBC	**Illegal:** no dining car after passenger cars
EEBBDC	**Illegal:** dining car after box car

Hint: my re pattern was 15 characters.

Syntax Diagram for Problem 7



Problem 7 Solution: train.py

```
# train.py
# problem 7:
# regular expression train
import re
# Sample strings.
list = ["EC","EEEPPDBBPDBBBBC","EEBB","EBBBC",
        "EEPPPPPPDBBC","EEPPBBC","EEBBDC"]
# Loop.
for element in list:
    # Match if two words starting with letter d.
                  #12345678901234567890
    m = re.match("^E+(P{1,4}D|BB)*C$", element)
    # See if success.
    if m:
        print(m.group(0))
```

Regular Expression Methods

Regex String Operations

LECTURE 1

Regular Expression Methods and Regex Pattern Object

Pattern Matching Methods:

- Generally, the functions discussed in this lecture operate on a **regular expression pattern** and **text**.
- These functions produce information related to attempting to match the pattern and text: which parts of the text matched which parts of the pattern.

Pattern Object (Re-usable Regex):

We can use the compile function to compile a pattern (producing a regex), and then call methods on that regex directly, as an object to perform the same operations as the functions, but more efficiently if the **pattern** is to be used repeatedly.

Note: We will omit discussing/using the [,flags] option in this discussion, but see section 6.2 of the Python Library Documentation for a discussion of A/ASCII, DEBUG, I/IGNORECASE, L/LOCALE, M/MULTILINE, S/DOTALL, and X/VERBOSE.

Regular Expression Functions

import re

re.match():

re.match(pattern, string[, flags])

Example:

match-object = re.match(pattern, string_to_be_matched)

Return Value:

- Returns a **match object**, consisting of tuple of groups (0, 1, ...)
- Return None if the string does not match the pattern; note that this is different from a zero-length match.
- Matches start at the text's beginning

Python program that uses match

Demo Program: re_match1.py

```
import re
# Sample strings.
list = ["dog dot", "do don't", "dumb-dumb", "no match"]
# Loop.
for element in list:
    # Match if two words starting with letter d.
    m = re.match("(d\w+)\W(d\w+)", element)
    # See if success.
    if m:
        print(m.groups())
```

Output

('dog', 'dot') ('do', 'don') ('dumb', 'dumb’)

Pattern details

- **Pattern: (d\w+)\W(d\w+)**
- **d** Lowercase letter d.
- **\w+** One or more word characters.
- **\W** A non-word character.

Python program that tests starts, ends

Demo Program: re_match2.py

```
import re
list = ["123", "4cat", "dog5", "6mouse"]
for element in list:
    # See if string starts in digit.
    m = re.match("^\d", element)
    if m: print("START:", element)
    # See if string ends in digit.
    m = re.match(".*\d$", element)
    if m: print(" END:", element)
```

Results

Output

```
START: 123
  END: 123
START: 4cat
  END: dog5
START: 6mouse
```

Pattern details

^\d	Match at the start, check for single digit.
.*\d$	Check for zero or more of any char.
	Check for single digit.
	Match at the end.

Python program that uses re, expressions, repeats, or

Demo Program: re_match3.py

```
import re
values = ["cat100", "---200", "xxxyyy", "jjj", "box4000", "tent500"]
for v in values:
    # Require 3 letters OR 3 dashes.
    # ... Also require 3 digits.
    m = re.match("(?:(?:\w{3})|(?:\-{3}))\d\d\d$", v)
    if m: print(" OK:", v)
    else: print("FAIL:", v)
```

Python program that uses re, expressions, repeats, or

Output	Pattern details	
OK: cat100	**(?:**	The start of a non-capturing group.
OK: ---200	**\w{3}**	Three word characters.
FAIL: xxxyyy	**|**	**Logical or:** a group within the chain must match.
FAIL: jjj		
FAIL: box4000	**\-**	An escaped hyphen.
FAIL: tent500	**\d**	A digit.
	$	The end of the string.

Regular Expression Functions

import re

re.search():

re.search(pattern, text [,flags])

Example:

match_object = re.search(pattern, string_to_be_searched)

Return Value:

- Returns a **match object**, consisting of tuple of only the first occurrence.
- Returns None is no match
- Matches can start anywhere in the text (different from re.match())

Python program that uses search

Demo Program: re_search1.py

```
import re
# Input.
value = "voorheesville"
m = re.search("(vi.*)", value)
if m: # This is reached.
    print("search:", m.group(1))
m = re.match("(vi.*)", value)
if m: # This is not reached.
    print("match:", m.group(1))
```

Output

```
search: ville
```

Pattern details

Pattern: (vi.*)

vi The lowercase letters v and i together.

.* Zero or more characters of any type.

Comparison between match() and search()

re.match():

re.match ("(a+)b","**aaab**") matches;

re.match ("(a+)b","xaaab")) doesn't match

re.search():

re.search("(a+)b","**aaab**") matches;

re.search("(a+)b","x**aaab**") matches

by using patterns like ^...$, these functions produce the same results

Regular Expression Functions

import re

re.findall():

re.findall(pattern, text [,flags])

Example:

- re.findall('a*b','**ab**a**ab**c**b**d**ab**c') returns ['ab', 'aab', 'b', 'ab']
- re.findall('((a*)(b))','abaabcbdabc') returns [('ab','a','b'), ('aab','aa','b'), ('b','','b'), ('ab','a','b')]

Return Value:

- Returns **a list of string**/of **tuples of string** (the groups), specifying matches
- Matches can start anywhere in the text;
- The next attempted match starts one character after the previous match terminates.
- If the pattern has groups, then the string matching each group is included in the resulting list too: use **?:** to avoid these groups

Python program that uses findall

Demo Program: re_findall.py

```
import re
# Input.
value = "abc 123 def 456 dot map pat"
# Find all words starting with d or p.
list = re.findall("[dp]\w+", value)
# Print result.
print(list)
```

Output

```
['def', 'dot', 'pat’]
```

Pattern details

Pattern: [dp]\w+

[dp] A lowercase d, or a lowercase p.

\w+ One or more word characters.

Regular Expression Functions

import re

re.findter():

re.finditer(pattern, text [,flags])

Example:

- iterable_object = re.finditer(pattern, string_to_be_operates)

Return Value:

- Returns iterable equivalent of findall
- Returns an iterator **yielding** match_object instances over all non-overlapping matches for the **re** pattern in the string.
- Need to be operated with **tuple(iterable_object.groups())** to have same result as **findall**

Python program that uses finditer

Demo Program: re_finditer.py

```
import re
value = "123 456 7890"
# Loop over all matches found.
for m in re.finditer("\d+", value):
    print(m.group(0))
    print("start index:", m.start())
```

Output

```
123
start index: 0
456
start index: 4
7890
start index: 8
```

Example for: re.finditer()
Demo Program:

```
>>> import re

>>> re.finditer(r'\w','http://www.hackerrank.com/')

<callable-iterator object at 0x0266C790>

>>> map(lambda x:
x.group(),re.finditer(r'\w','http://www.hackerrank.com/'))

['h', 't', 't', 'p', 'w', 'w', 'w', 'h', 'a', 'c', 'k', 'e', 'r', 'r', 'a', 'n', 'k', 'c', 'o',
'm']
```

Regular Expression Functions

import re

re.split():

re.split(pattern, text [,maxsplit, flags])

Example:

list_of_strings = re.split(pattern, string_to_be_split)

Return Value:

- Returns **a list of strings**: much like calling **text.split(...)**

Example for re.split()

- **re.split(pattern, text [,maxsplit, flags])** like the text.split(...) method, but using a regular expressions pattern to determine how to split the text:
- **re.split('\.|-', 'a.b-c')** returns ['a','b','c’], spliting on either

  . (which must be written here as \.) or

  - which standard string split function, **text.split(...)** can't do;

Note that 'a.b-c'.split(".-") splits only on '.-' both a . followed by a -, so in this case it fails to split anywhere, since '.-' is not anywhere in the text at all.

- If the pattern has groups, then the text matching each group is included in the resulting list too: use **?:** to avoid these groups.

Example for re.split()

Examples:

re.split(';+' ,'abc;d;;e') returns ['abc', 'd', 'e']

re.split('(;+)','abc;d;;e') returns ['abc', ';', 'd', ';;', 'e']

Python program that uses split

Demo Program: re_split1.py

```
import re
# Input string.
value = "one 1 two 2 three 3"
# Separate on one or more non-digit characters.
result = re.split("\D+", value)
# Print results.
for element in result:
    print(element)
```

Output

```
1
2
3
```

Pattern details

Pattern: \D+

\D+ One or more non-digit characters.

Regular Expression Functions

import re

re.sub():

re.sub(pattern, repl, text, [,count, flags])

Example:

- re.sub('(a+)','"as"','aabcaaadaf’) returns "as"bc"as"d"as"f
- re.sub('(a+)','(\g<1>)','aabcaaadaf’) returns (aa)bc(aaa)d(a)f

Return Value:

- Returns a string

Explanation: re.sub()

re.sub(pattern, repl, text, [,count, flags])

- in text, replace pattern by **repl** (which may refer to matched groups via \# (e.g. \1) or \g<#>, (e.g., \g<1>), or \g<name>
- **repl:** a replacement string or a **callable** (function, lambda) return string
- (where name comes from ?P<name>) or a function that is passed a match object);
- if there is no match, then it just returns the text parameter's value, unchanged

Python program that uses string replacement

Demo Program: re_sub1.py

```
import re
# An example string.
v = "running eating reading"
# Replace words starting with "r" and ending in "ing"
# ... with a new string.    *? Means .* using lazy algorithm
v = re.sub(r"r.*?ing", "ring", v)
print(v)
```

Output

```
ring eating ring
```

Python program that uses re.sub

Demo Program: re_sub2.py

```
import re
def multiply(m):
    # Convert group 0 to an integer.
    v = int(m.group(0))
    # Multiply integer by 2.
    # ... Convert back into string and return it.
    return str(v * 2)
# Use pattern of 1 or more digits.
# ... Use multiply method as second argument.
result = re.sub("\d+", multiply, "10 20 30 40 50")
print(result)
```

Output

```
20 40 60 80 100
```

Python program that uses re.sub, lambda

Demo Program: re_sub3.py

```
import re
# The input string.
input = "laugh eat sleep think"
# Use lambda to add "ing" to all words.
result = re.sub("\w+", lambda m: m.group(0) + "ing", input)
# Display result.
print(result)
```

Output

```
laughing eating sleeping thinking
```

Python program that uses re.sub with dictionary

Demo Program: re_sub4.py

```
import re
plants = {"flower": 1, "tree": 1, "grass": 1}
def modify(m):
    v = m.group(0)
    # If string is in dictionary, return different string.
    if v in plants:
        return "PLANT"
    # Do not change anything.
    return v
# Modify to remove all strings within the dictionary.
result = re.sub("\w+", modify, "bird flower dog fish tree")
print(result)
```

Output

bird PLANT dog fish PLANT

Regular Expression Functions

import re

re.subn():

re.subn(pattern, repl, text, [,count, flags])

Example:

- re.subn

Return Value:

- same as sub but returns a **tuple**: (new string, number of subs made)

Python program that calls re.subn

Demo Program: re_subn.py

```
import re
def add(m):
    # Convert.
    v = int(m.group(0))
    # Add 2.
    return str(v + 1)
# Call re.subn.
result = re.subn("\d+", add, "1 2 3 4 5")
print("Result string:", result[0])
print("Number of substitutions:", result[1])
```

Output

```
Result string: 2 3 4 5 6
Number of substitutions: 5
```

Regular Expression Functions

import re

re.escape():

re.escape(string)

Example:

- re.escape('^a.*$’)

Return Value:

- Returns a string
- Return string with all non-alphanumerics backslashed; this is useful if you want to match an arbitrary literal string that may have regular expression metacharacters in it.

Non-Overlapping Matching

- In **findall** and **sub/subn**, only non-overlapping patterns are found/replaced:
- in text **aaaa** there are two non-overlapping occurrence of the pattern **aa**: starting in index 0 and 2 (not in index 1, which overlaps with the previous match in indexes **0-1**).

Regular Expression Methods

Regex Pattern Object and Operations

LECTURE 1

Regular Expression Functions

import re

re.compile():

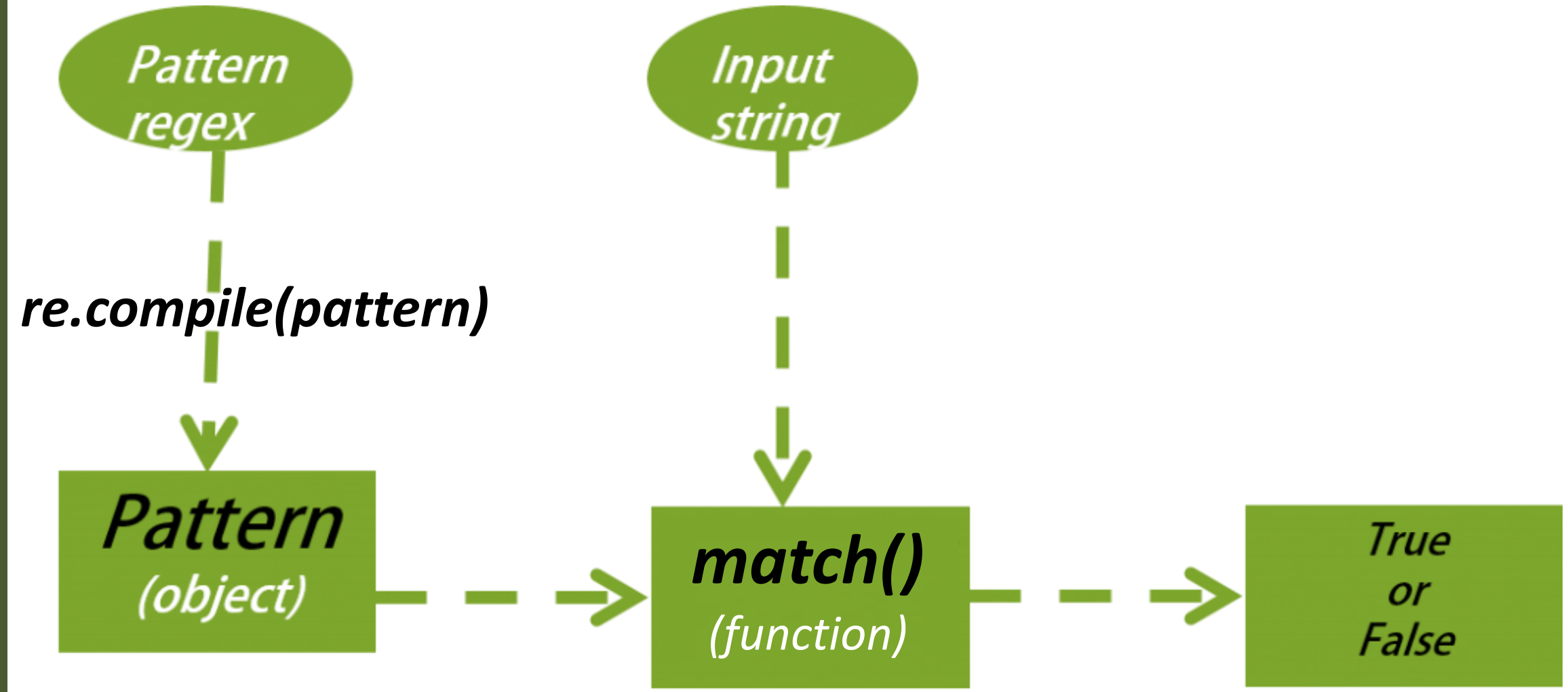
compile(pattern, [,flags])

Example:

compiled_regex_pattern_object = re.compile(pattern)

Return Value:

- Returns a regex (compiled pattern) object.


Pattern
regex
Input
string
re.compile(pattern)
Pattern
(object)
match()
(function)
True
or
False

Regex

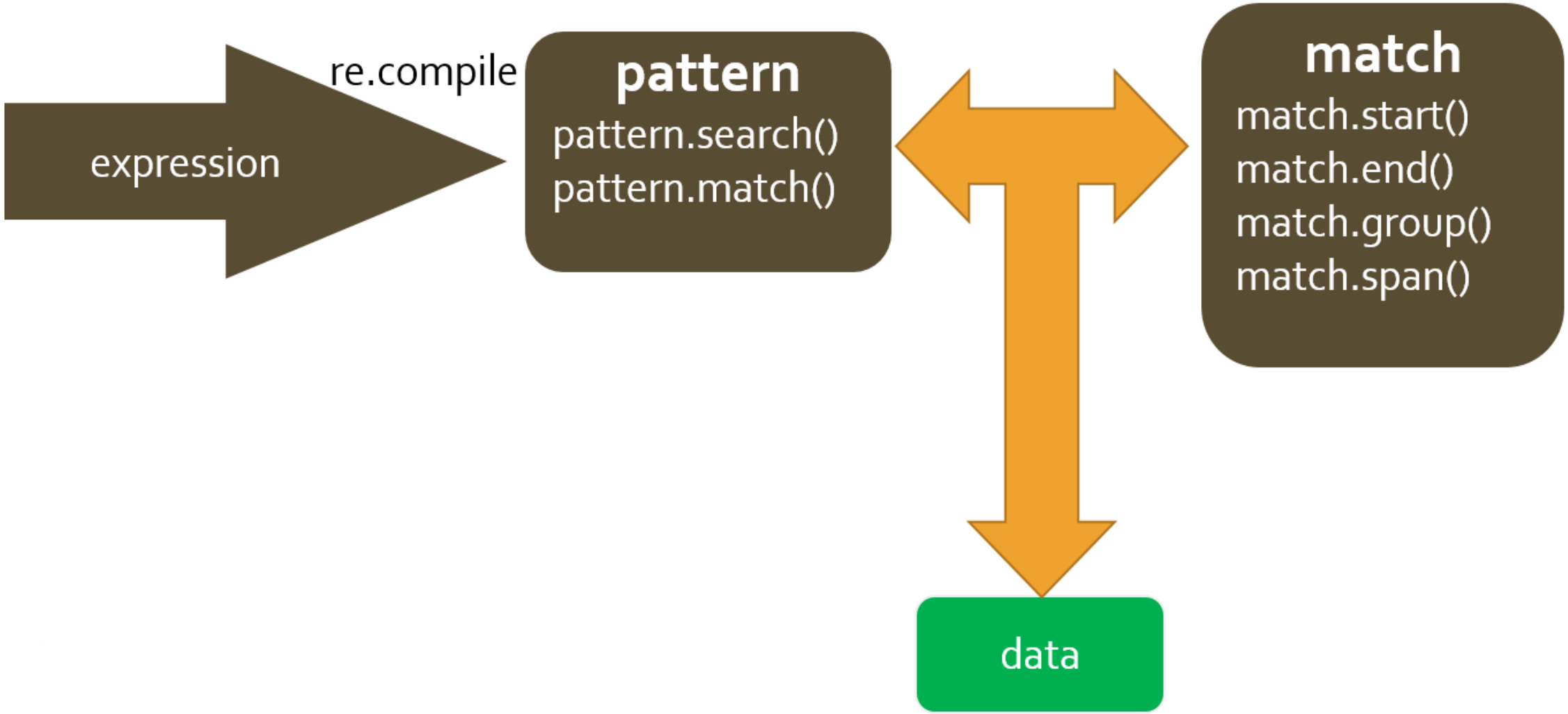
- regex (compiled pattern) object methods (see the compile method above, which produces regexes) are called like

  c = re.compile(p)

- It is then efficient to call **c.match(...)** many times.
- Calling **re.match(p,...)** many times with the same pattern recompiles and matches the pattern each time re.match is called; whereas **c = re.compile(p)** compiles the pattern once and **c.match(...)** matches it each time it is called.

Regex

- Using this feature allows us to compile a pattern and reuse it for all the operations above: **re.match(p,s)** is just like
  re.compile(p).match(s)
- if we are doing many matches with the same pattern, compile the pattern once and use it with the match method below many times.
- **pos**/**endpos** are options that specify where in text the match starts and ends (from **pos** to **endpos**-1).
- **pos** defaults to **0** (the beginning of the text) and
- **endpos** defaults to the length of the text so **endpos-1** is its last character).


expression
re.compile
pattern
pattern.search()
pattern.match()
match
match.start()
match.end()
match.group()
match.span()
data

Instance Methods for Pattern Objects

Each of the re functions above has an equivalent method using a compiled pattern to call the method, but omitting the pattern from its argument list.

match(text [,pos][,endpos])	See match above, with pos/endpos
search(text [,pos][,endpos])	See search above, with pos/endpos
findall(text [,pos][,endpos])	See findall above, with pos/endpos
finditer(text [,pos][,endpos])	See finditer above, with pos/endpos
split (text [,maxsplit])	See split above, with pos/endpos
sub(repl, text [,count])	See sub above, with pos/endpos
subn(repl, text [,count])	See subn above, with pos/endpos

Re-writing of Python Programs with Regex

- So, for example, instead of writing

```
for line in open_file:
  ...re.match(pattern_string,line)
```

- which implicitly compiles the same **pattern_string** during each loop iteration (whenever **re.match** executes) we can write

```
for line in open_file:
  pattern = re.compile(pattern_string)
  ...pattern.match(line)
```

Re-writing of Python Programs with Regex

- which explicitly compiles the **pattern_string** during each loop and uses the compiled version (insted of the function re.match) to call "**match**" (just two ways of doing the same thing). If we know the **pattern_string** stays the same,
- we can also write

```
pattern = re.compile(pattern_string)
for line in open_file:
  ...pattern.match(line)
```

Re-writing of Python Programs with Regex

- which explicitly compiles the **pattern_string** ONCE, before the loop executes, and calls "**match**" on it during each loop iteration.
- See the **grep.py** module in the **remethods** download that accompanies this lecture for code that calls **re.compile**.

Match Objects and Groups

LECTURE 1

Match Objects

Match objects record information about which parts of a pattern match the text. **Each group** (referred to by either its number or an optional name) can be used as an argument to a function that specifies information about the **start**, **end**, **span**, and **characters** in the matching text.

- Calling **match/search** produces **None** or a **match object**
- Calling **findall** produces **None**, a **list of strings** (if there are no groups) or **a list of tuples of strings** (if there are groups, with the tuple index representing the each group #)
- Calling **finditer** produces **None** or an **iterable of groups** (not used in the course)

Groups

1. Each **group** is indexed by a **number** or **name** (a name only when the group was delimited by (?P<name>)); **group 0** is all the character in the match, **groups 1-n** are for delimited matches inside.
   - For example, in the pattern (a)(b(c)(d)) the a is in group 1, the b is in group 2, c is in group 3, and d in is group 4: groups are numbered by in what order we reach their OPENING parenthesis. Technically, group 2 includes all of b(c)(d), the characters in groups 2-4.
2. Note that if a parenthesized expression looks like (**?:**...) it is NOT numbered as a group. So in (a)(**?:**b(c)(d)) the a is in group 1, the b is in NO group, c is in group 2, and d in is group 3.

Groups

- If a group is followed by a **?** and the pattern in the group is skipped, its group will be **None**.
- In the result of **re.match('a(b)?c','ac')** group 1 will be **None**. If the group itself is not optional, but the text inside the group is, the group will show as matching an empty string.
- So the result of re.match('a(b?)c','ac') group 1 will be **" (Empty String)**.
- The same is true for a repetition that matches 0 times. Compare re.match('a(b)*c','ac') group 1 and re.match('a(b*)c','ac') group 1.

Groups

- If a group matches multiple times (e.g., a(.)*c), only its **last match** is available, so for axyzc group 1, the (.) group, is bound to the character z. If we wrote this as a(.*)c the (.*) group is bound to the characters **xyz**. If we wrote it as a((.)*)c **group 1** is **xyz** and **groups 2** is just **z**.
- Printing the groups of match object prints a tuple of the matching characters for each group 1-n (not group #0)

Match Objects' Methods

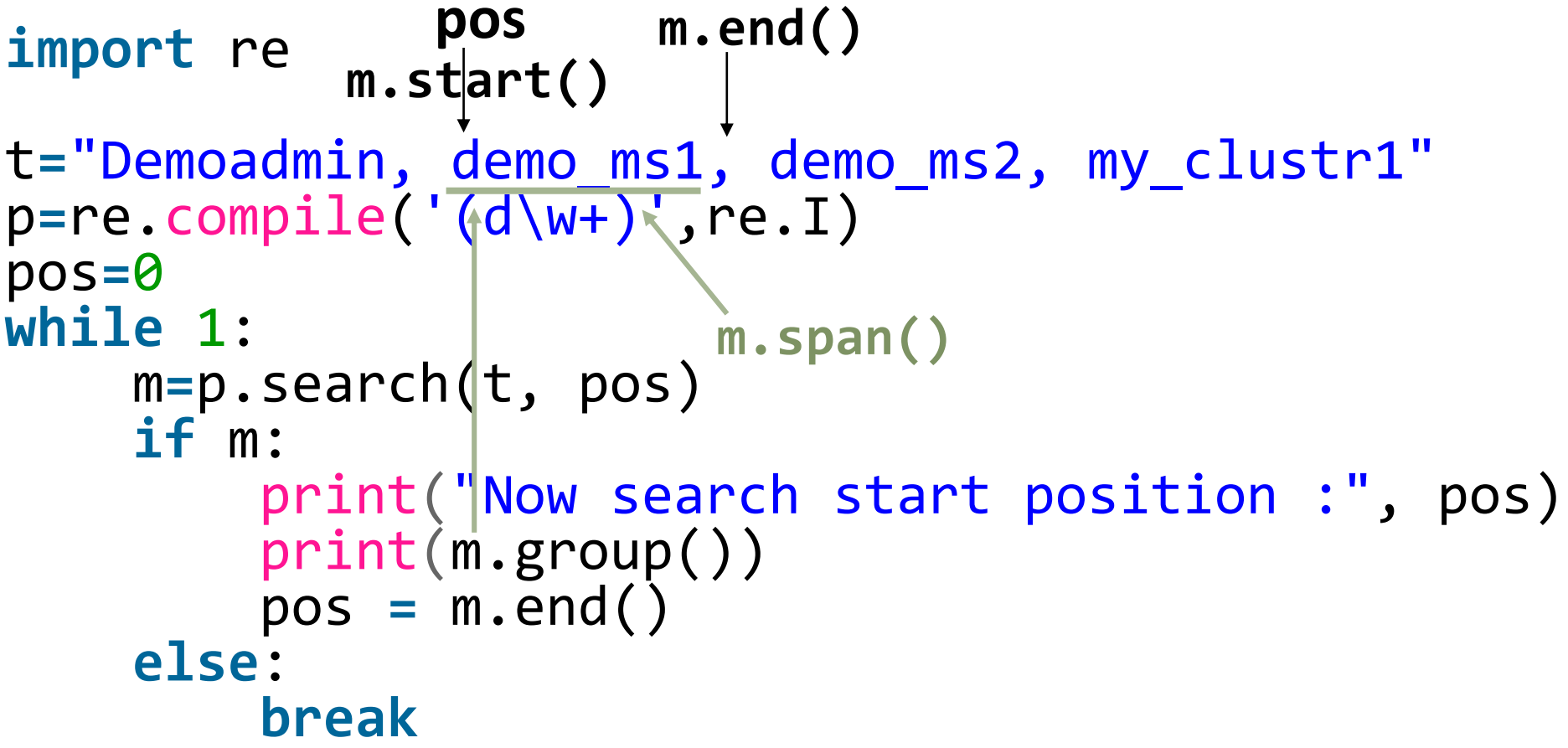
LECTURE 1

Match Object's Methods

- We can look at each resulting group by its number (including group #0), using any of the following methods that operate on match objects
  - **group(g)** text of group with specified name or number
  - **group(g1,g2, …)** tuple of text of groups with specified name or number
  - **groups()** tuple of text of all groups (can iterate over tuple)
  - **groupdict()** text of all groups as dict (see ?P<name>)
  - **start([group])** starting index of group (or entire matched string)
  - **end([group])** ending index of group (or entire matched string)
  - **span([group])** tuple: (start([group]), end([group]))
- Try doing some matches and calling .groups() on the result.

Group is a way of iterating through the string

Demo Program: compile.py




```
import re

t="Demoadmin, demo_ms1, demo_ms2, my_clustr1"
p=re.compile('(d\w+)',re.I)
pos=0
while 1:
    m=p.search(t, pos)
    if m:
        print("Now search start position :", pos)
        print(m.group())
        pos = m.end()
    else:
        break
```

Regular expression FLAGS

re.I == re.IGNORECASE Ignore case

re.L == re.LOCALE Make \w, \b, and \s locale dependent

re.M == re.MULTILINE Multiline

re.S == re.DOTALL Dot matches all (including newline)

re.U == re.UNICODE Make \w, \b, \d, and \s unicode dependent

re.X == re.VERBOSE Verbose (unescaped whitespace in pattern is ignored, and '#' marks comment lines)

Packages for Example

Unzip **remethods.zip** and examine the phonecall.py and readingtest.py modules for examples of Python programs that use regular expressions (and groups) to perform useful computations.

Python program that uses groupdict

Demo Program: groupdict.py

```
import re
name = "Roberta Alden"
# Match names.
m = re.match("(?P<first>\w+)\W+(?P<last>\w+)", name)
if m: # Get dict. d is a list for dict {‘first’:’Roberta’, ‘last’:’Alden’}
    d = m.groupdict()
    # Loop over dictionary with for-loop.
    for t in d:
        print(" key:", t)
        print("value:", d[t])
```

Output

```
key: last
value: Alden
key: first
value: Roberta
```

Python program that uses Regex comments (?#)

Demo Program: group1.py

```
import re
data = "bird frog"
# Use comments inside a regular expression.
m = re.match("(?#Before part).+?(?#Separator)\W(?#End part)(.+)", data)
if m: print(m.group(1))
```

Output

```
frog
```

Pattern details

(?#Before part)	Comment, ignored
.+?	As few characters as possible
(?#Separator)	Comment, ignored
\W	Non-word character
(?#End part)	Comment, ignored
(.+)	One or more characters, captured

Python program that uses not-followed-by pattern (?!)

Demo Program: pattern1.py

```
import re
data = "100cat 200cat 300dog 400cat 500car"
# Find all 3-digit strings except those followed by "dog" string.
# ... Dogs are not allowed.
m = re.findall("(?!\d\d\ddog)(\d\d\d)", data)
print(m)
```

Output

['100', '200', '400', '500']

Pattern details

(?!\d\d\ddog)	Not followed by 3 digits and "dog"
(\d\d\d)	3 digit value

A Simple but Illustrative Example

LECTURE 1

Demo Program: phone.py

```
import re
phone = r'^(?:\((\d{3})\))?(\d{3})[-.](\d{4})$'
m = re.match(phone,'(949)824-2704')
assert m != None, 'No match'
print(m.groups())
area, exchange, number = [int(i) if i != None else None for i in m.group(1,2,3)]
print(area, exchange, number)
```

Regular Expression Example:

1) Here, phone is a pattern anchored at both ends.

(a) It starts with ^(?:**\((\d{3})\)**)?...

- controlling an optional area code.
- The **?:** means that the parentheses are not used to create a group, but are used with the **?** (option) symbol.
- Inside it is **\((\d{3})\)**: a left parenthesis, group 1 which consists of any 3 digits, and a right parenthesis.

(b) Next is **(\d{3})** group 2, which consists of any 3 digits.

(c) Next is **[-.]** that is one symbol, either a - or . (not in a group).

(d) Next is **(\d{4})** group 3, which consists of any 4 digits.

Regular Expression Example:

2) Calling the re.match function matches the pattern against some text, it returns a match object that is bound to m.

3) If the match m is **None**, there is no match (raises **AssertionError** exception).

4) Converts every non-None string from groups 1, 2, and 3 into an int.

Regular Expression Example:

5) Prints the the groups

Try also replacing line 2 by

```
m = re.match(phone,'824-2704’)                  # area is None
m = re.match(phone,'(949)824.2704')             # . instead of -; no match
m = re.match(phone,'(94)824-2704')              # only 2 in area code; no match
```

Also, we can replace the first two lines by the following equivalent lines

```
phone_pat = re.compile(r'^(?:\((\d{3})\))?(\d{3})[-.](\d{4})$')
m = phone_pat.match('(949)824-2704')
```

Extra Topics

LECTURE 1

Raw Strings

Make Sure Escape Sequence Don’t Happen for Regular Expression Pattern Strings

When writing regular expression pattern strings as arguments in Python it is best to use raw strings: they are written in the form **r'...'** or **r"..."**.

These should be used because of an issue dealing with using the backslash character in patterns, which is sometimes necessary.

For example, in regular strings when you write **'\n'** Python turns that into a **1** character string with the newline character: **len('\n')** is ***1***. But with raw strings, writing r'\n' specifies a string with a backslash followed by an n: **len(r'\n')** is **2**. Normally this isn’t a big issue because writing '\d' or' *' in regular strings doesn't generate an escape character, since there is no escape character for **d** or **(** so len('\d') and len('*') is 2.

**d in function/method calls

(where d is a dict, variable parameter list)

- If we call a function we can specify **d as one or more of its arguments. For each **d, Python uses all its keys as parameter names and all its values as default arguments for these parameter names. For example

  f(**{'b':2, 'a':1, 'c':3}) is translated by Python into f(b=2,a=1,c=3)

- Note that this is useful in regular expressions if we use the (?P<name> ...) option and then the groupdict() method for the match it produces.
- There is also a version that works the other way. Suppose we have a functions
- whose header is

```
def f(x,y,**kargs):  # The typical name is **kargs
```

**d in function/method calls

(where d is a dict)

- if we call it by f(1,2,a=3,b=4,c=5) then

  x is bound to 1

  y is bound to 2

  kargs is bound to a dictionary {'b':4, 'a':3, 'c':5}

- See the argument/parameter matching rules from the review lecturer for a complete description of what happens.
- So (in reverse of the order explained above) ** as a parameter creates a dictionary of "extra" named-arguments suppled when the function is called, and ** as an argument supplies lots of named-arguments to the function call. We will cover this information again when we examine inheritance
- The parse_phone_named method (in phoncecall.py) uses this language feature.

Translation of a Regular Expression Pattern into a NDFA

- How do the functions/methods in re compile a regular expression and match it against text? It translates every regular expression into a non-deterministic finite automaton (see Programming Assignment #1), and then matches against the text (ibid) to see if the match succeeds (reaches the special last state).
- The general algorithm (known as **Thompson's Algorithm**) is a bit beyond the scope of this course and uses a concept we haven't discussed (epsilon transitions), but you can look up the details if you are intereseted. Here is an example for the regular expression pattern **((a*|b)cd)+.** It produces an **NDFA** described by

Translation of a Regular Expression Pattern into a NDFA

start;a;1;a;2;b;2;c;3

1;a;1;a;2

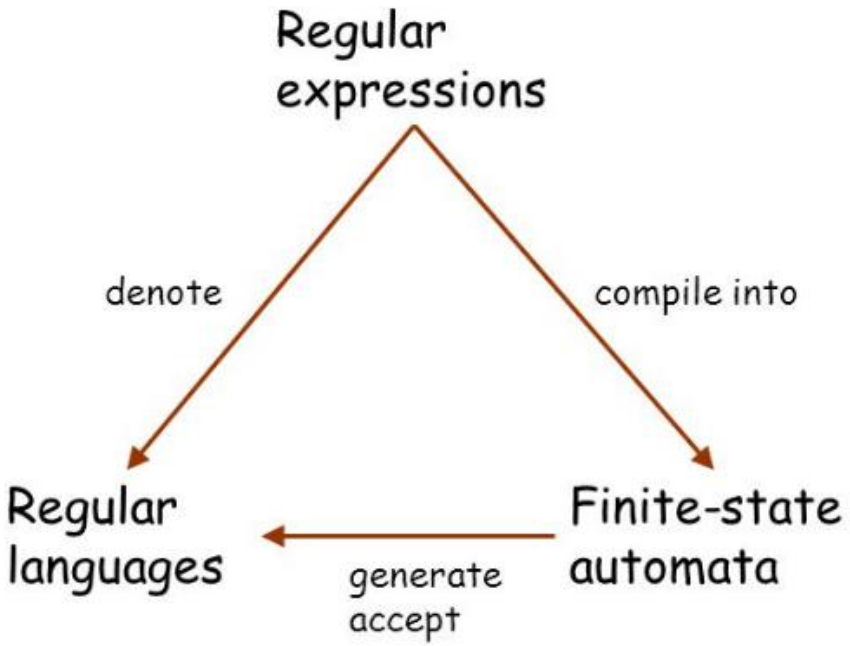
2;c;3

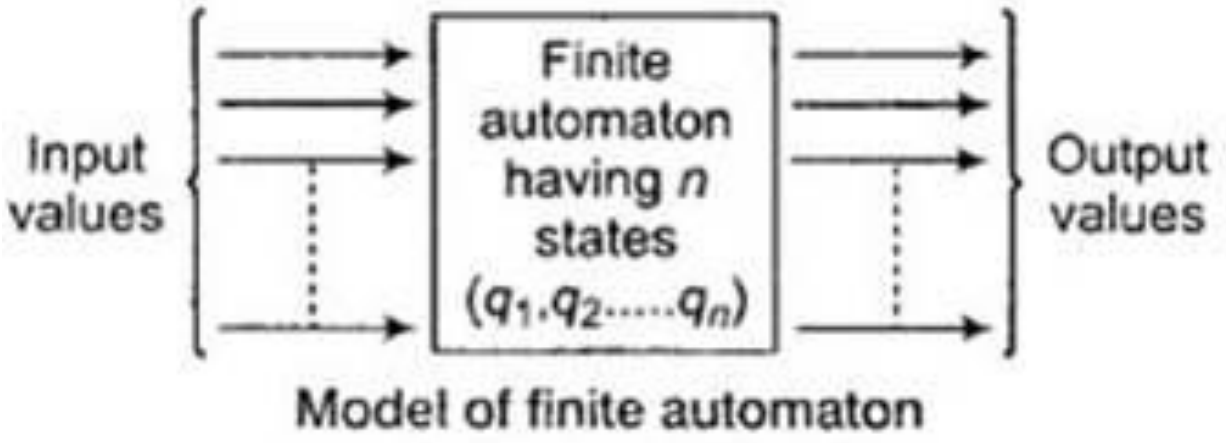
3;d;start;d;last

last

- This pattern matches a text string by starting in state **'start'** and exhausting all the characters and having **'last'** in its possible states.

Regular Expression, Regular Languages, Finite State Automata, and Finite State Machine

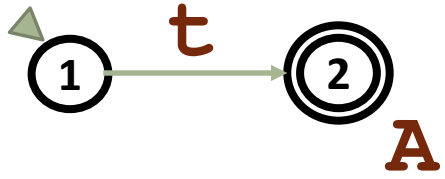
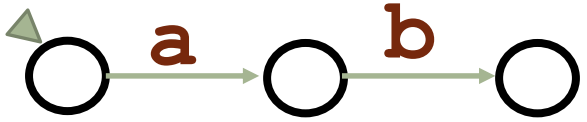
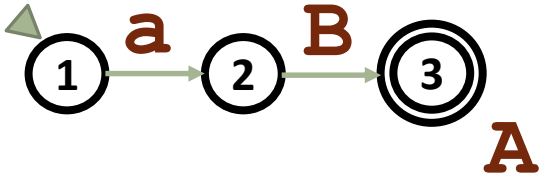
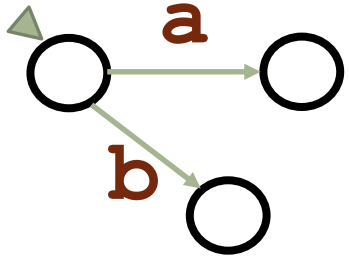
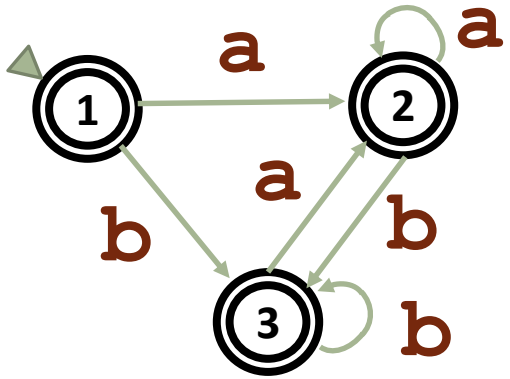




Model of finite automaton

Conversion of Regular Expression to Finite State Automaton

Starting state

Regular Expression	Finite Automaton	Regular Grammar Rules	Finite Automaton
a		A -> t	
ab		A -> aB	
a|b		A -> (a|b)*	
a*			

Regular Expression Designs (Part 2)

LECTURE 1

Problems:

Write functions using regular expression patterns

8. Write a function named **contract** that takes a string as a parameter.

It substitutes the word 'goal' to replace any occurrences of variants of this word written with any number of o's, e.g., 'gooooal') in its argument. So calling contract('It is a goooooal! A gooal.') returns 'It is a goal! A goal.'.

Problem 8 Solution: goal.py

```
# goal.py
# string substitue (replace)
import re

def contract(v):
    v = re.sub(r"go+al", "goal", v)
    return v

v = contract("goal gooal goooooooal gooooooooal goooooooooal”+
             ” gooal!! goooooooal!! gooal!!!")
print(v)
```

Problem 8 Solution: goal2.py

```
# goal2.py
# string substitue (replace)
import re

def contract(v):
    p = re.compile(r"go+al")
    v = p.sub("goal", v)
    return v

v = contract("goal gooal goooooooal gooooooooal goooooooooal”+
             ” gooal!! goooooooal!! gooal!!!")
print(v)
```

Problems:

Write functions using regular expression patterns

9. Write a function named **grep** that takes a regular expression pattern string and a file name as parameters.

It returns a list of 3-tuples consisting of the file-name, line number, and line of the file, for each line whose text matches the pattern.

Hint: Using enumerate and a comprehension, this is a 3 line function, but you can use explicit looping in a longer function.

Problem 9 Solution: grep.py

```
# grep.py
import re
def grep(filename, pattern):
    list=[]
    i = 1
    for line in open(filename, 'r').readlines():
        line = line.rstrip()
        m = pattern.match(line)
        if m: list.append((filename, i, line))
        i += 1
    return list
def main():
    filename = "data.txt"
    pattern = re.compile("(0|-?[1-9]\d*)")
    list = grep(filename, pattern)
    for f, lineno, st in list:
        print("%-10s Line %d %s" % (f, lineno, st))
if __name__ == "__main__":
    main()
```

Problems:

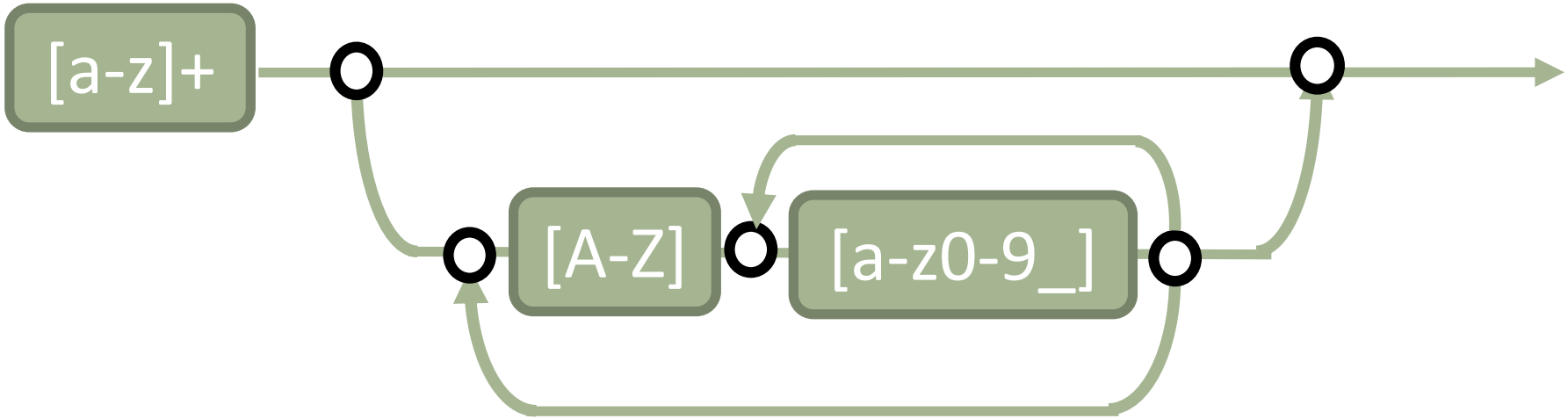
Write functions using regular expression patterns

10. Write a function named **name_convert** that takes two file names as parameter.

It reads the first file (which should be a Python program) and writes each line into the second file, but with identifiers originally written in camel notation converted to underscore notation:

e.g. **aCamelName** converts to **a_camel_name**. **Camel** identifiers start with a lower-case letter followed by upper/lower-case letters and digits: each upper-case letter is preceded by an underscore and turned into a lower-case letter.

Java ID pattern



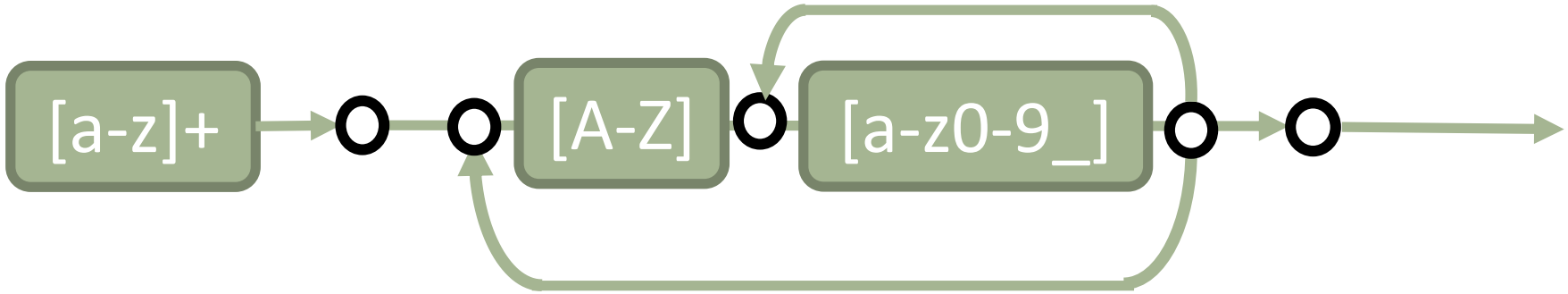
Java Variable (id) Convention:

Lower case for learing word, Uppercase for other words.

variableForStudentGrade

variable

Java ID pattern (Words need Replacement)



Java Variables (id) those Need Conversion:

Lower case for learing word, Uppercase for other words.

variableForStudentGrade

Pseudo Code

1. Open a text file
2. For each line in the text file.

        replace the tokens that matches the Java ID Regular

            Expression Pattern

        add the line to a list.
3. close the text file.
4. Write all of the lines in the list back to the text file.

Demo Program 1: java_match.py

Design the Java id matching regular expression:

```
# java_match.py
import re
list = ["javaVariable","computer
science","variableA", "floatingNumber",
"initialVariableValue"]
for element in list:
      m = re.match("[a-z]+([A-Z][a-z0-9_]*)*",
element)
      if m:
            print(m.group(0))
```

Demo Program 2: java_iteration.py

Use Regular Expression to find the Characters to be replaced:

```
def main():
    name_convert("Loan.java", "Loan2.java")

if __name__ == "__main__":
    main()
```

Demo Program 2: java_iteration.py

```
# java_replacement.py
import re
def calculate_new_string(token): # replacement of C by _c
    ch_list = []
    for j in range(len(token)):
        if (token[j].isupper() and j!=0):
            ch_list.append(token[j])
    for ch in ch_list:
        token = re.sub(ch, "_" + ch.lower(), token)
    return token
```

```
def name_convert(filename1, filename2):
    f1 = open(filename1, "r")
    p = re.compile("[a-z]+([A-Z][a-z0-9_]*)+")
    newlines = []
    for line in f1.readlines():
        replacement_list = []
        new_list = []
        pos = 0
        m = p.search(line, pos)
        while m:  # find all id's which need to be modifiedd
            pos = m.end()
            replacement_list.append(m.group())
            new_list.append(calculate_new_string(m.group()))
            m = p.search(line, pos)
        for pair in list(zip(replacement_list, new_list)):
            line = re.sub(pair[0], pair[1], line)
        newlines.append(line)
    f1.close()
    f2 = open(filename2, "w")
    for line in newlines:
        f2.write(line)
    f2.close()
```

Java Style Variable Name Regex Pattern

List to hold lines after conversion

Search for a new match

While the line has new matches

The id to be replaced

The id after conversion

Zip the old and new string together and make it iterable as list

Convert the line

Problem 10: Conversion of Variable Names from Java Style to C/C++ (Python) Style

// Loan.java (part)

```java
public class Loan {
  private double annualInterestRate;
  private int numberOfYears;
  private double loanAmount;
  private java.util.Date loanDate;

  /** Default constructor */
  public Loan() {
    this(2.5, 1, 1000);
  }

  /** Construct a loan with specified annual interest rate,
      number of years, and loan amount
   */
  public Loan(double annualInterestRate, int numberOfYears,
      double loanAmount) {
    this.annualInterestRate = annualInterestRate;
    this.numberOfYears = numberOfYears;
    this.loanAmount = loanAmount;
    loanDate = new java.util.Date();
  }
}
```

// Loan2.java (part)

```java
public class Loan {
  private double annual_interest_rate;
  private int number_of_years;
  private double loan_amount;
  private java.util.Date loan_date;

  /** Default constructor */
  public Loan() {
    this(2.5, 1, 1000);
  }

  /** Construct a loan with specified annual interest rate,
      number of years, and loan amount
   */
  public Loan(double annual_interest_rate, int number_of_years,
      double loan_amount) {
    this.annual_interest_rate = annual_interest_rate;
    this.number_of_years = number_of_years;
    this.loan_amount = loan_amount;
    loan_date = new java.util.Date();
  }
}
```