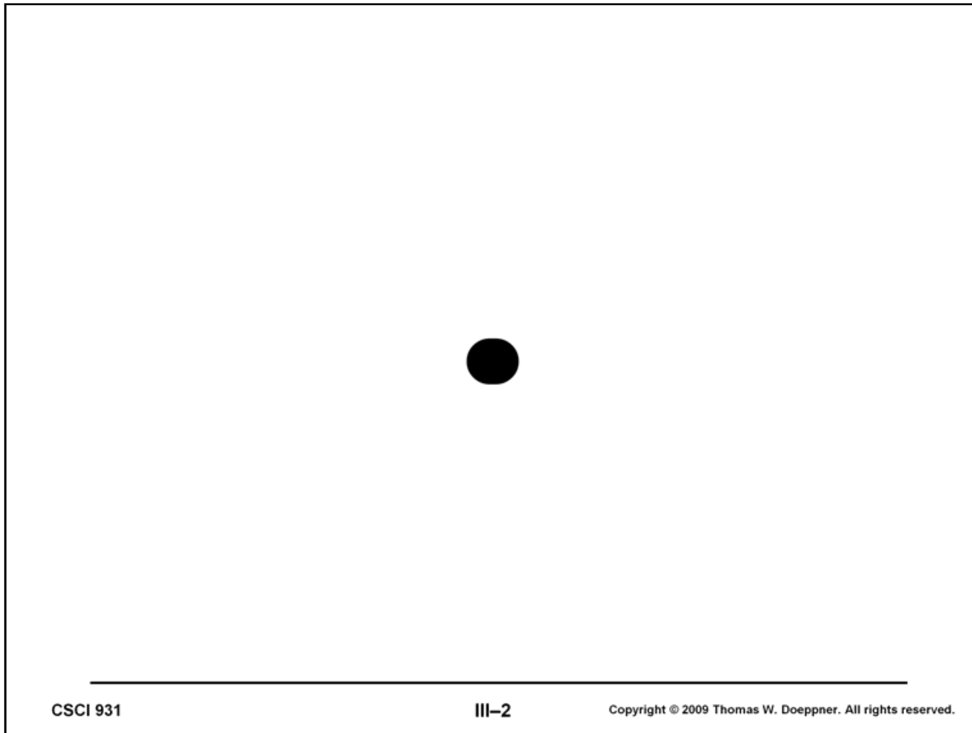
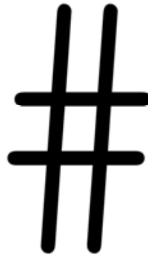
CS 931: More Python

Copyright © 2009 Thomas W. Doeppner. All rights reserved.



Period, as a regular expression, matches any character.



 Copyright © 2009 Thomas W. Doeppner. All rights reserved.

Any characters after (and including) a pound sign in a line of text are ignored. This is useful for providing comments in your code, explaining what it is the code is supposed to be doing. Comments are extremely important. This is not only so that other people can figure out what you've done, but also so that you, looking at your own code next week, can figure out what you've done.

 Copyright © 2009 Thomas W. Doeppner. All rights reserved.

"r" introduces a "raw" character string. In such strings Python treats characters following back slashes as normal characters, but does not remove the back slashes. For example, `'\n'` is a string containing just one character, the newline character. Whereas `r'\n'` is a string containing two characters, back slash and lower-case n. The string `'\''` consists of one character, a single quote, while `r'\''` consists of two characters, a backslash and a single quote. The string `r'\'` is a syntax error, since the second single quote doesn't delimit a string (and thus there is no single quote to match the first).

Raw strings are particularly important with regular expressions. For example, \b, as a regular expression, matches a word boundary. But \b also is the way to write the backspace character. Thus the string `'\ba\b'`, when used as a regular expression, matches the backspace character, followed by a lower-case a, followed by a backspace character. Once could write `'\\ba\\b'` to get a regular expression that matches the word consisting of the single character a. But the same effect as achieved with `r'\ba\b'`.

Match Objects

```
>>> import re
>>> re.match('(\w+) +(\w+)', 'Shriram Krishnamurthi, computer
  scientist')
<_sre.SRE_Match object at 0x02A93F98>
>>> x=re.match('(\w+) +(\w+)', 'Shriram Krishnamurthi, computer
  scientist')
>>> x.group(0)
'Shriram Krishnamurthi'
>>> x.group(1)
'Shriram'
>>> x.group(2)
'Krishnamurthi'
```

 Copyright © 2009 Thomas W. Doeppner. All rights reserved.

Some Python functions that deal with regular expressions, such as re.match and re.search, return “match objects”. These are thingies (a technical term) that represent what the regular expression has matched. As shown in the slide, if *x* is a match object, then *x.group(0)* produces what was matched. But note that if the regular expression contains parenthesized terms, as in the slide, then *x.group(1)* produces what was matched by the first parenthesized term, *x.group(2)* produces what was matched by the second parenthesized term, and so forth.

Truth Defined …

- What's not false is true
  - 0 is false
  - any other integer is true

  - '' is false
  - any other string is true

  - [] is false
  - any other list is true

  - None is false
  - any other object is true

CSCI 931 III–6 Copyright © 2009 Thomas W. Doeppner. All rights reserved.

Note that `re.match` and `re.search` return `None` if they don't find a match, otherwise they return match objects. Thus their results can be used in *if* statements.

Name Scopes

```
>>> x = 6
>>> y = 7
>>> def func(a, b):
        x = a + b + y
        return x

>>> a = 2
>>> b = 3
>>> z = func(b, a)
>>> w = func(x, y)
```

scope 1

scope 2

scope 1 (continued)

Copyright © 2009 Thomas W. Doeppner. All rights reserved.

A *scope* is a part of a program in which identical names refer to the same things. Function definitions are unique scopes. Thus the slide shows two scopes, with the function defining a new scope, while the original scope (1) continues after the end of the function. We think of scope 1 as being the *outer scope* and scope 2 as being the *inner scope*. The two references to y in scope 1 refer to the same thing. The references to a and b in the two scopes refer to different things in each scope. What does y refer to in scope 2? It turns out that, in this case, it refers to the same thing that y refers to in scope 1. The rule is that if a name is used just to look at something within a scope, but nothing has yet been assigned to that name, then the name refers to the same thing it does in the outer scope (if there is one — otherwise it's an error). However, if a name is assigned to, such as x in scope 2, then the name becomes distinct from the same name used in the outer scope. Thus x in scope 2 is distinct from x in scope 1.

Name Scopes (more)

```
>>> x = 6
>>> y = 7
>>> def func(a, b):
        global x
        x = a + b + y
        return x

>>> a = 2
>>> b = 3
>>> z = func(b, a)
>>> w = func(x, y)
```

 Copyright © 2009 Thomas W. Doeppner. All rights reserved.

Here we've inserted a *global* statement in scope 2. It indicates that the name following it is the same as the name that appears in the outer scope. Thus assigning to x within the function modifies the x in the outer scope.

Name Spaces

```
>>> import RE1 # getbook is defined in RE1.py
>>> def getbook(s):
    return 'got book ' + s

>>> b1 = getbook('MobyDick.txt')
>>> b2 = RE1.getbook('MobyDick.txt')
```

 Copyright © 2009 Thomas W. Doeppner. All rights reserved.

The first reference to *getbook* is to the one defined in the current scope. If there were no *getbook* in the scope, then the statement would fail.

The second reference to *getbook* explicitly refers to the RE1 module, which has its own separate name space.