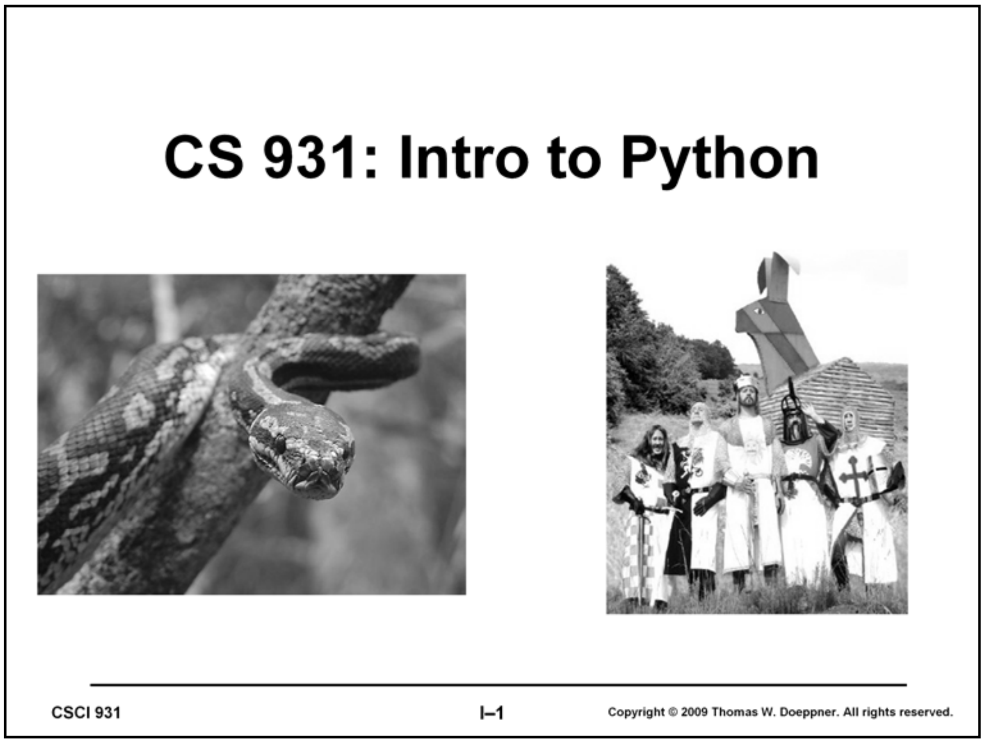
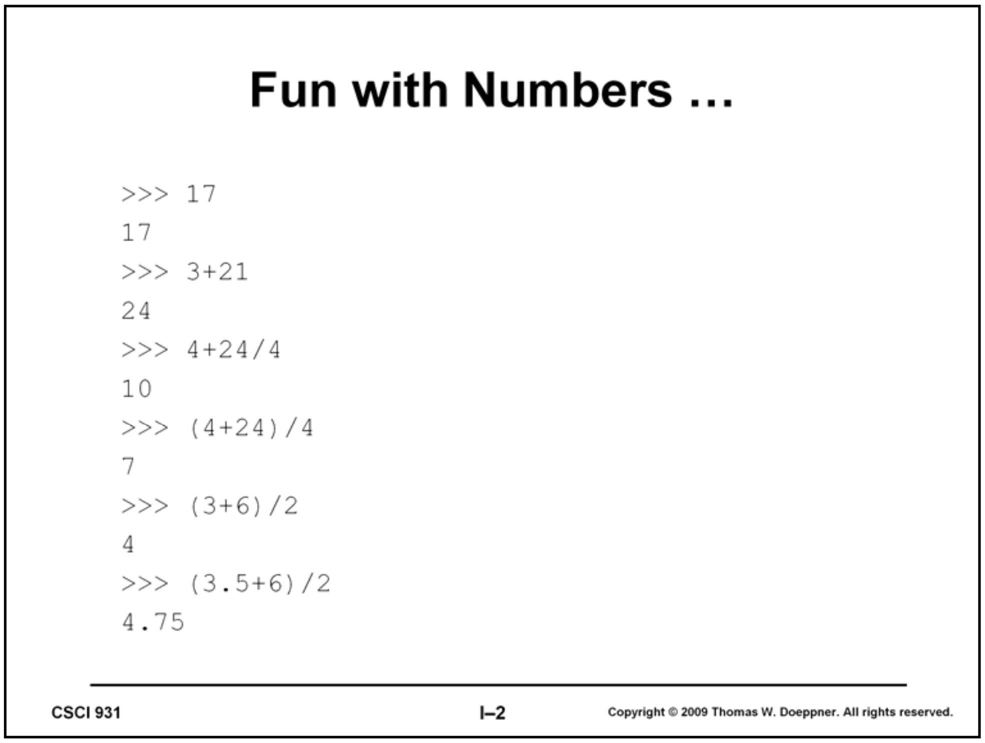


CS 931: Intro to Python

CSCI 931 I–1

Python, the programming language, was actually named after the Monty Python troupe. A fairly readable (at least by computer scientists …) tutorial on Python can be found at http://docs.python.org/tutorial/. If you would like to download Python onto your own computer, go to http://www.python.org/download/. The version that we're using in this course is 2.6.3. (Why aren't we using the most recent version, 3.1.1? It's because they've made some changes that are incompatible with some existing Python programs that we intend to use.)



Fun with Numbers …

```
>>> 17
17
>>> 3+21
24
>>> 4+24/4
10
>>> (4+24)/4
7
>>> (3+6)/2
4
>>> (3.5+6)/2
4.75
```

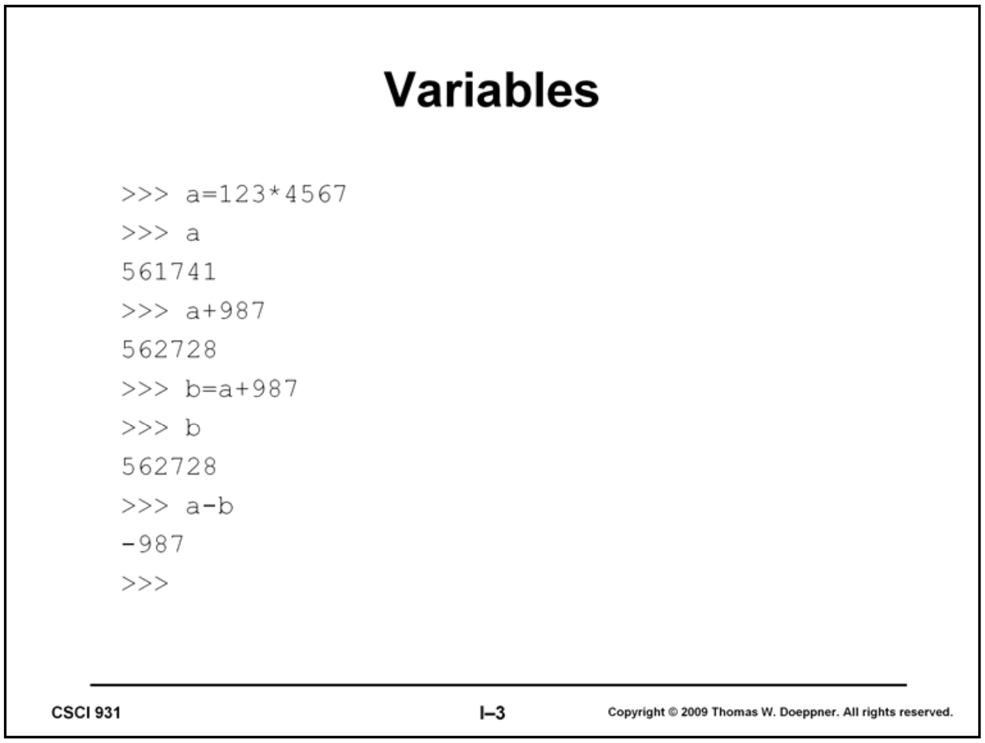
CSCI 931 I–2 Copyright © 2009 Thomas W. Doeppner. All rights reserved.

In addition to the Python programming language, there is a "Python programming environment" in which you write and run Python programs. There are actually at least two such environments. The one we use in this course is called IDLE (Integrated DeveLopment Environment for python). When you run the IDLE program, a window pops up and within it you are given a prompt: ">>>". This is Python's way of telling you that the next move is up to you, i.e., you should type something.

For starters, we use Python as a desktop calculator. If you type in any form of arithmetic expression, it responds with its value. So, the value of 17 is, (tough one …) 17. "3+21" is pretty obvious, but what about "4+24/4"? Here we get into issues of the *precedence* of *operators*. Things like "+", "-", "/", and "*" (multiplication) are operators. However, "/" and "*" have higher precedence than "+" and "-". Thus in the expression "4+24/4", the operator "/" is used before the operator "="; thus first Python computes 24/4, then it adds 4 to the result. If you want to do the addition first, then you indicate this with the use of parentheses.

You might think that the value of "(3+6)/2" should be 4.5. However, unless you force it to think otherwise, Python deals with integers. Thus it first computes 3+6, then divides the result by 2. When you divide 9 by 2, you get an integer quotient (4) and a remainder (1). Since Python wants the answer to be an integer, it throws away the remainder and uses the integer quotient as the result.

To convince Python that you really would like it to use numbers with decimals, you have to include at least one such number in the computation. Thus in the expression "(3.5+6)/2", the 3.5 tells Python that you are prepared to deal with non-integers, and thus it gives you a result with decimals.

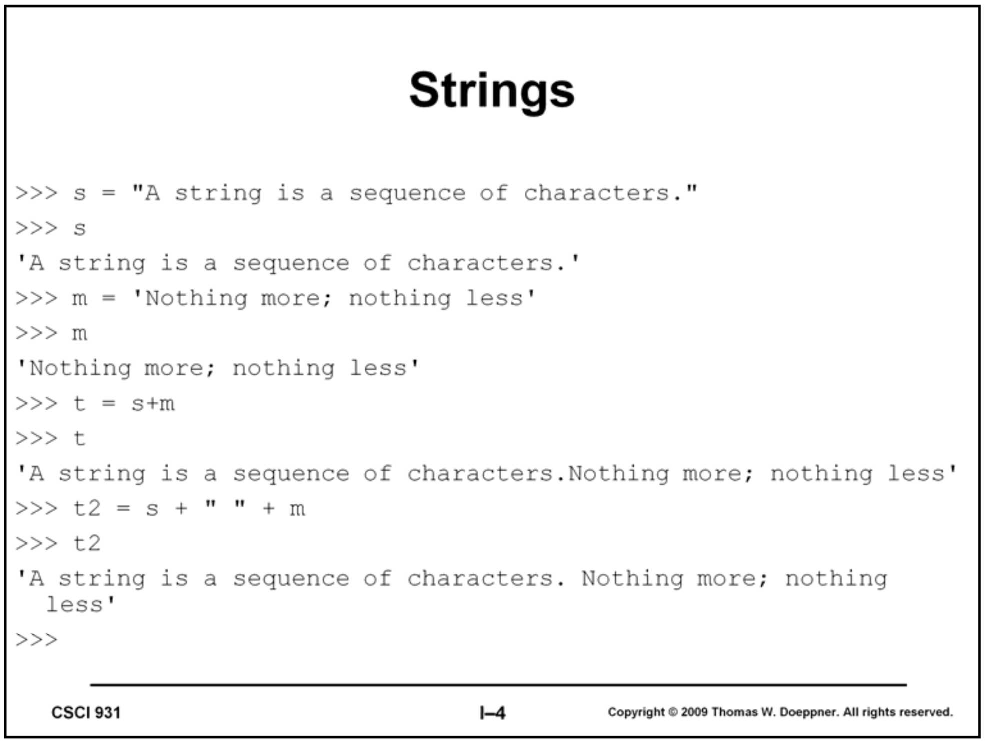


Variables

```
>>> a=123*4567
>>> a
561741
>>> a+987
562728
>>> b=a+987
>>> b
562728
>>> a-b
-987
>>>
```

CSCI 931 I–3 Copyright © 2009 Thomas W. Doeppner. All rights reserved.

It really makes things a lot easier if you can use identifiers, such as *a*, *b*, or even *value1* to refer to values of various sorts. Thus in the first line of this slide, we let *a* refer to the value of the expression 123*4567. Python will tell us what such an identifier refers to if we type its name into IDLE. Such identifiers are commonly called "variables," since their values can vary (depending on what we do to them).

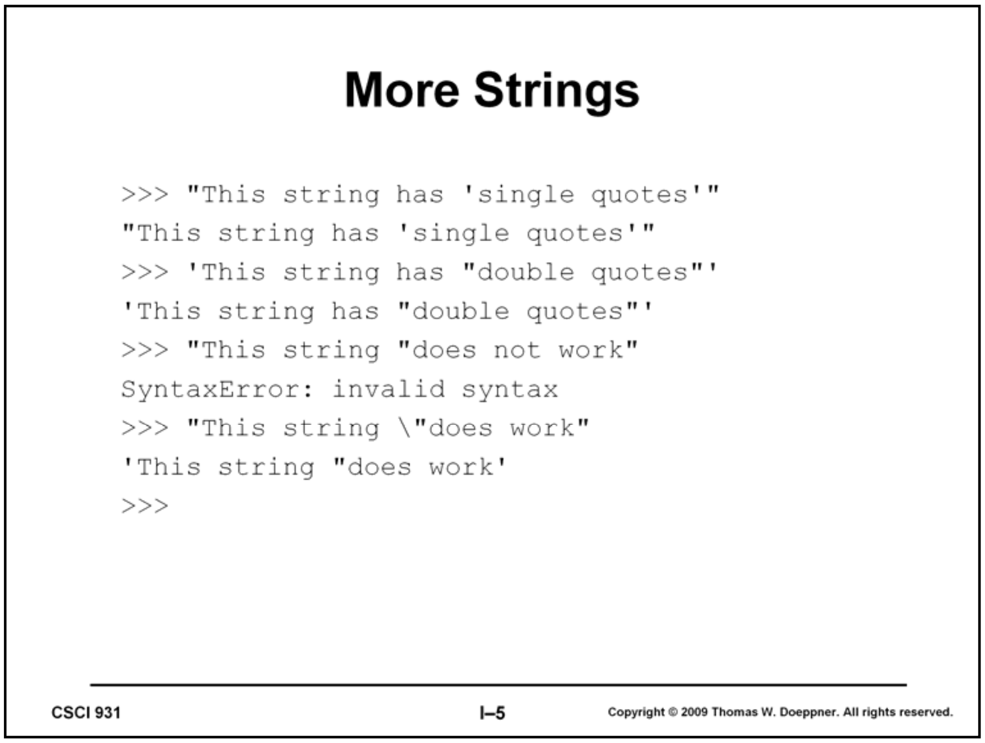


Strings

```
>>> s = "A string is a sequence of characters."
>>> s
'A string is a sequence of characters.'
>>> m = 'Nothing more; nothing less'
>>> m
'Nothing more; nothing less'
>>> t = s+m
>>> t
'A string is a sequence of characters.Nothing more; nothing less'
>>> t2 = s + " " + m
>>> t2
'A string is a sequence of characters. Nothing more; nothing
  less'
>>>
```

CSCI 931 I–4 Copyright © 2009 Thomas W. Doeppner. All rights reserved.

Python is equally adept at working with strings of characters as it is at working with numbers. In the first line of the slide we've set *s* to refer to such a string. Note that strings are enclosed in either double quotes or single quotes. The "+" operator, when applied to strings, produces a new string that is the result of connecting together its two operands.

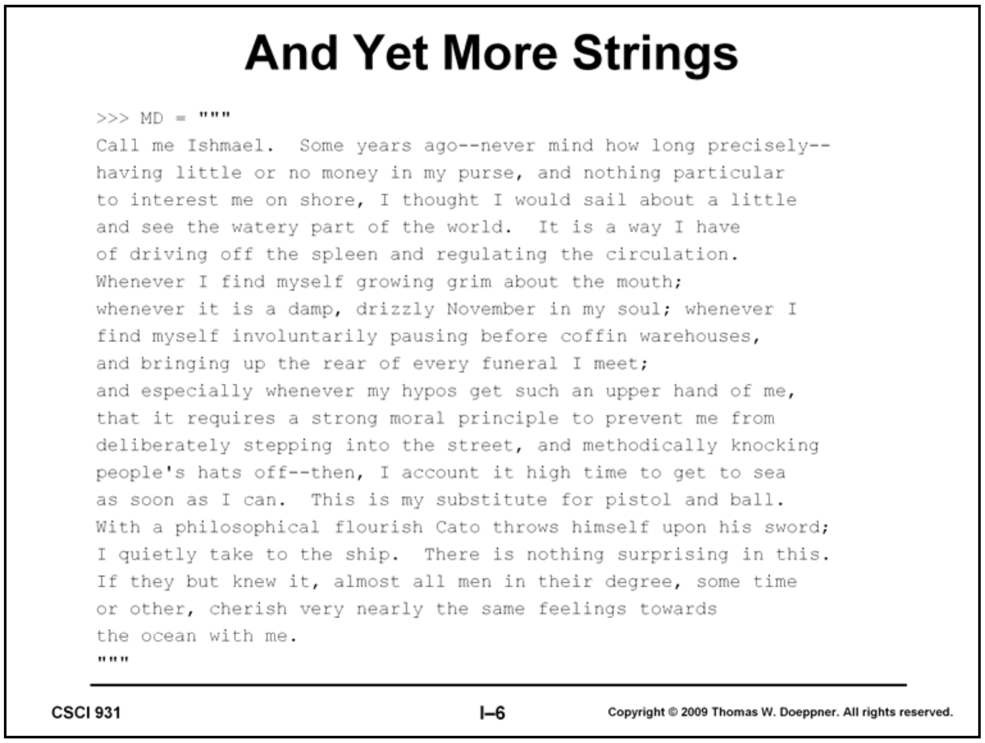


More Strings

```
>>> "This string has 'single quotes'"
"This string has 'single quotes'"
>>> 'This string has "double quotes"'
'This string has "double quotes"'
>>> "This string "does not work"
SyntaxError: invalid syntax
>>> "This string \"does work"
'This string "does work'
>>>
```

CSCI 931 I–5

If a string is enclosed in double quotes, then single quotes may appear inside as part of the string (and vice versa). You can also tell Python to use a double quote or a single quote as a normal character (i.e., not as something that delineates a string) by putting before it a reverse slash (\).

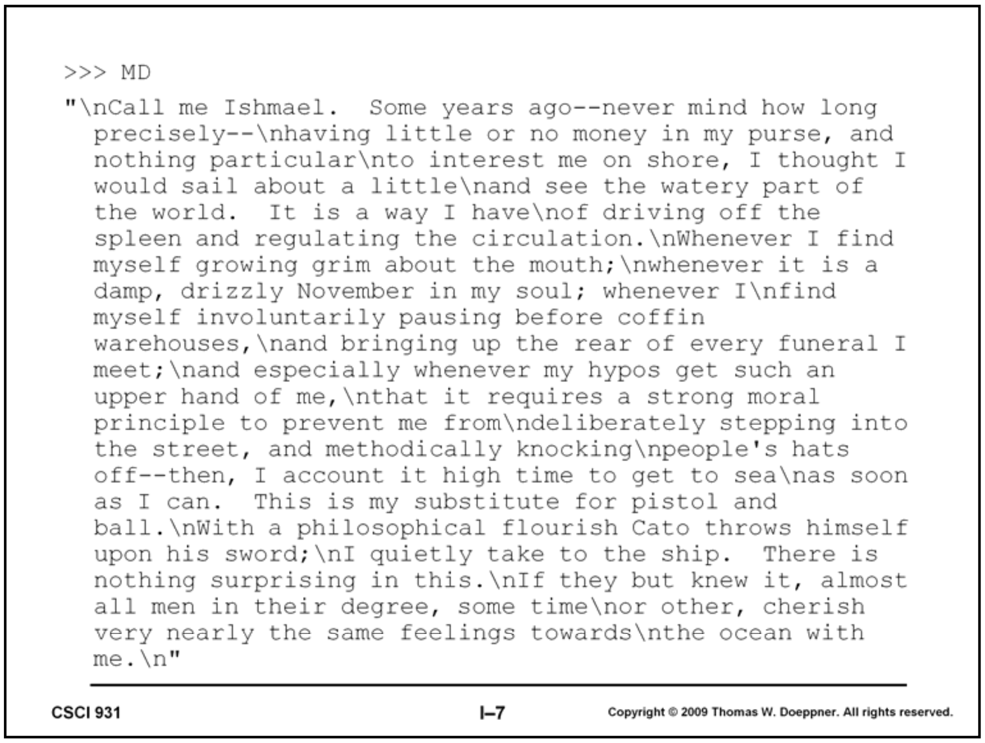


And Yet More Strings

```
>>> MD = """
Call me Ishmael.  Some years ago--never mind how long precisely--
having little or no money in my purse, and nothing particular
to interest me on shore, I thought I would sail about a little
and see the watery part of the world.  It is a way I have
of driving off the spleen and regulating the circulation.
Whenever I find myself growing grim about the mouth;
whenever it is a damp, drizzly November in my soul; whenever I
find myself involuntarily pausing before coffin warehouses,
and bringing up the rear of every funeral I meet;
and especially whenever my hypos get such an upper hand of me,
that it requires a strong moral principle to prevent me from
deliberately stepping into the street, and methodically knocking
people's hats off--then, I account it high time to get to sea
as soon as I can.  This is my substitute for pistol and ball.
With a philosophical flourish Cato throws himself upon his sword;
I quietly take to the ship.  There is nothing surprising in this.
If they but knew it, almost all men in their degree, some time
or other, cherish very nearly the same feelings towards
the ocean with me.
"""
```

CSCI 931 I–6

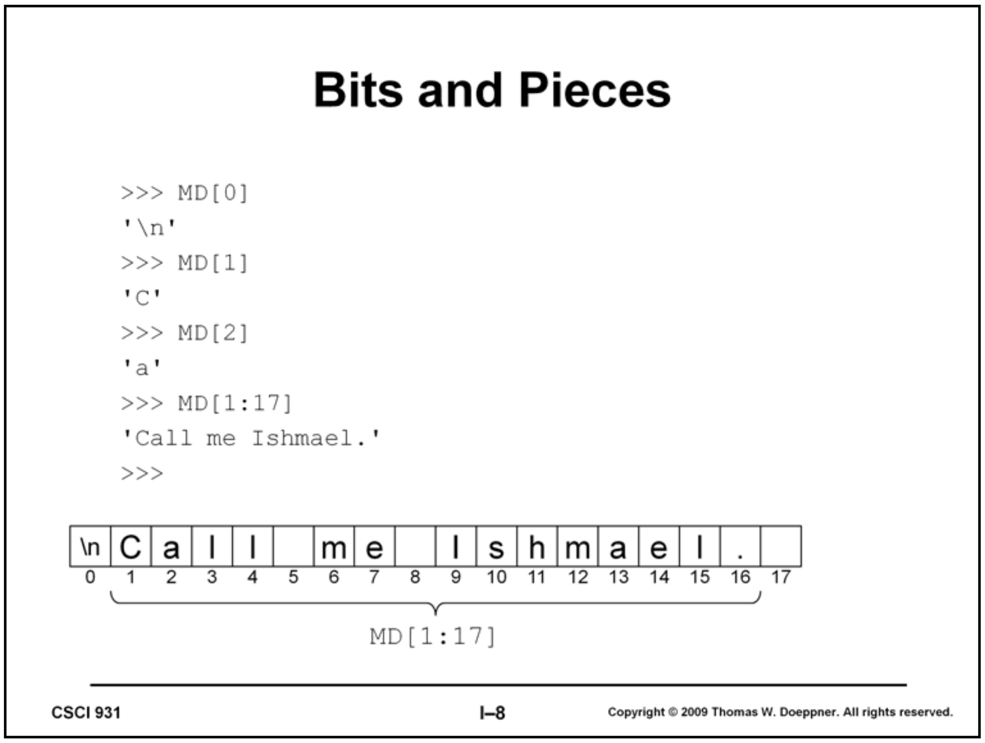
Normally, Python expects each line of text to contain one distinct thing, where “thing” includes arithmetic expressions and strings of characters. If you’ve got a really long string, this isn’t all that convenient. So, a long, multi-line, string of characters can be delineated with triple double quotes, as shown in the slide.



```
>>> MD
"\nCall me Ishmael.  Some years ago--never mind how long
  precisely--\nhaving little or no money in my purse, and
  nothing particular\nto interest me on shore, I thought I
  would sail about a little\nand see the watery part of
  the world.  It is a way I have\nof driving off the
  spleen and regulating the circulation.\nWhenever I find
  myself growing grim about the mouth;\nwhenever it is a
  damp, drizzly November in my soul; whenever I\nfind
  myself involuntarily pausing before coffin
  warehouses,\nand bringing up the rear of every funeral I
  meet;\nand especially whenever my hypos get such an
  upper hand of me,\nthat it requires a strong moral
  principle to prevent me from\ndeliberately stepping into
  the street, and methodically knocking\npeople's hats
  off--then, I account it high time to get to sea\nas soon
  as I can.  This is my substitute for pistol and
  ball.\nWith a philosophical flourish Cato throws himself
  upon his sword;\nI quietly take to the ship.  There is
  nothing surprising in this.\nIf they but knew it, almost
  all men in their degree, some time\nor other, cherish
  very nearly the same feelings towards\nthe ocean with
  me.\n"
```

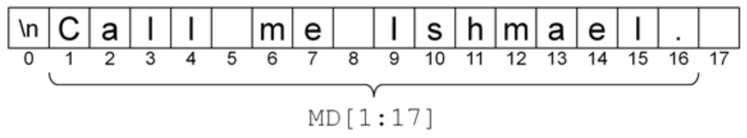
CSCI 931 I–7 Copyright © 2009 Thomas W. Doeppner. All rights reserved.

However, by using triple double quotes to delineate a long string, the breaks between lines become part of the string of characters. So, after setting MD to refer to multi-line string in the previous slide, when we print its value here, we get one really long string. The original line breaks have been replace with "\n". This two-character combination actually represents a single character called the *newline* character (sometimes referred to as the *linefeed* character). The long string referred to by MD is too long to fit on one line, so it is "wrapped" to fit within the width of the window.



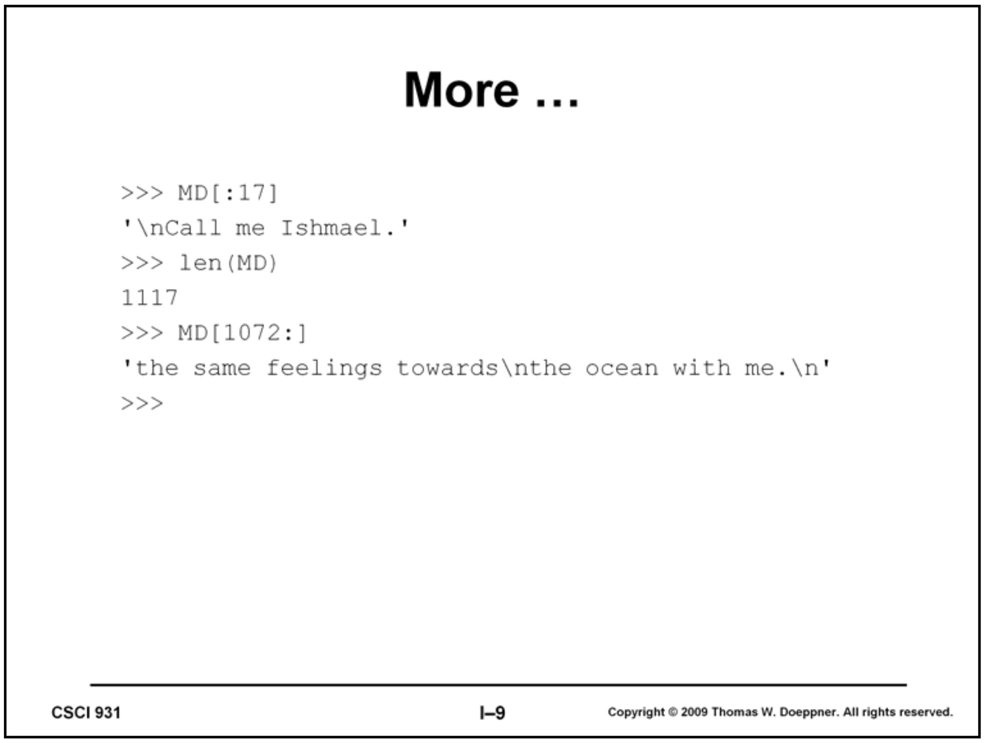
Bits and Pieces

```
>>> MD[0]
'\n'
>>> MD[1]
'C'
>>> MD[2]
'a'
>>> MD[1:17]
'Call me Ishmael.'
>>>
```



CSCI 931 I–8 Copyright © 2009 Thomas W. Doeppner. All rights reserved.

Given a string, Python allows us to refer to various pieces of it. The first character of the string referred to by *MD* is itself referred to by *MD[0].* (Why 0 and not 1? It's because this first character is zero characters beyond the beginning of the string. This is important for computers, less so for people.) An expression such as *MD[1:17]* refers to a "slice" of the string. This slice begins just before *MD[1]* and ends just before *MD[17].* This isn't necessarily the most obvious way of interpreting this, but, as we will see, it actually makes sense.



More …

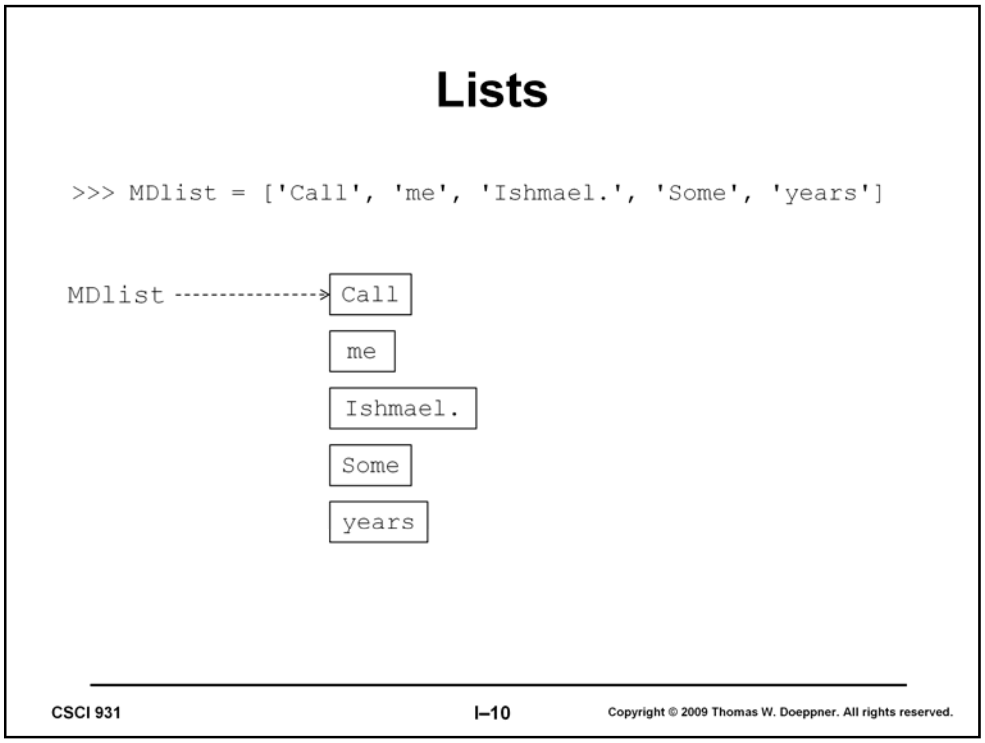
```
>>> MD[:17]
'\nCall me Ishmael.'
>>> len(MD)
1117
>>> MD[1072:]
'the same feelings towards\nthe ocean with me.\n'
>>>
```

CSCI 931 I–9 Copyright © 2009 Thomas W. Doeppner. All rights reserved.

The slide shows some shorthand for working with slices. If you leave off the number before the colon, then it's as if you had put a zero before the colon. This may not seem like you've saved a whole lot of typing, but it makes it clear as to how long the slice is (it's the number just after the colon).

The slide next shows the use of "len", which is a Python function. We'll see numerous examples of functions; with this one you follow it with a string in parentheses, and its value is the length of that string.

Finally, we see what happens if you leave out the number following the colon: its as if you had instead typed in the length of the string. Thus *MD[1072:]* means the slice starting at *MD[1072]* and continuing to the end of the string.

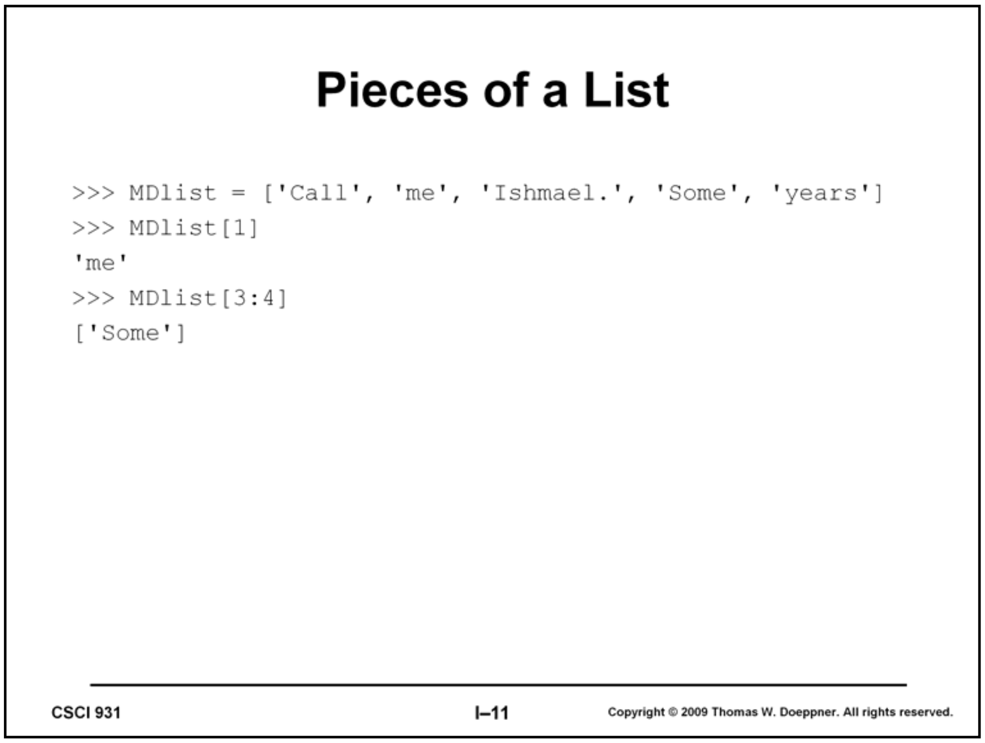


Lists

```
>>> MDlist = ['Call', 'me', 'Ishmael.', 'Some', 'years']
```

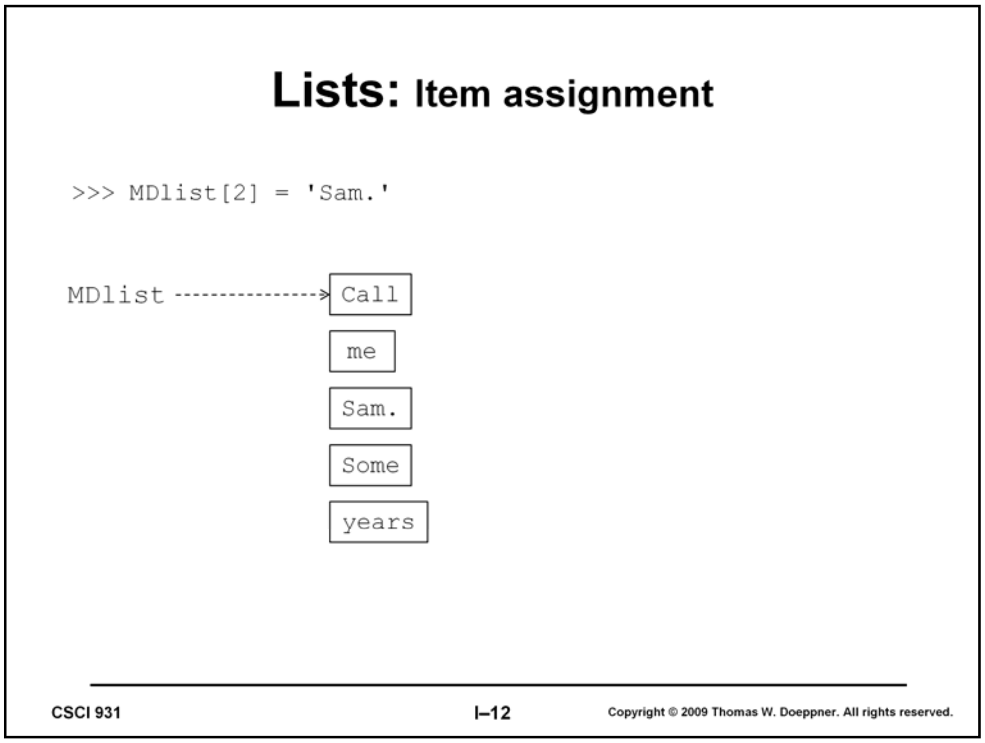
MDlist
Call
me
Ishmael.
Some
years

A string is a sequence of characters. Often that's what we want, but sometimes what we want is a sequence of something more complicated. For example, if our interest is not in characters but in words, we might want a sequence of words rather than of characters. You might think this isn't a whole lot different from a sequence of characters, but it allows us to refer to individual words rather than to characters.



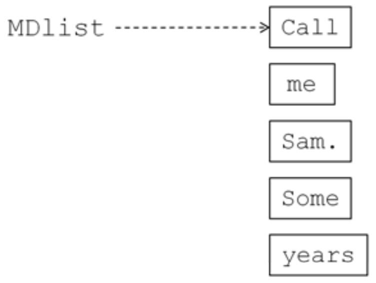
Pieces of a List

```
>>> MDlist = ['Call', 'me', 'Ishmael.', 'Some', 'years']
>>> MDlist[1]
'me'
>>> MDlist[3:4]
['Some']
```



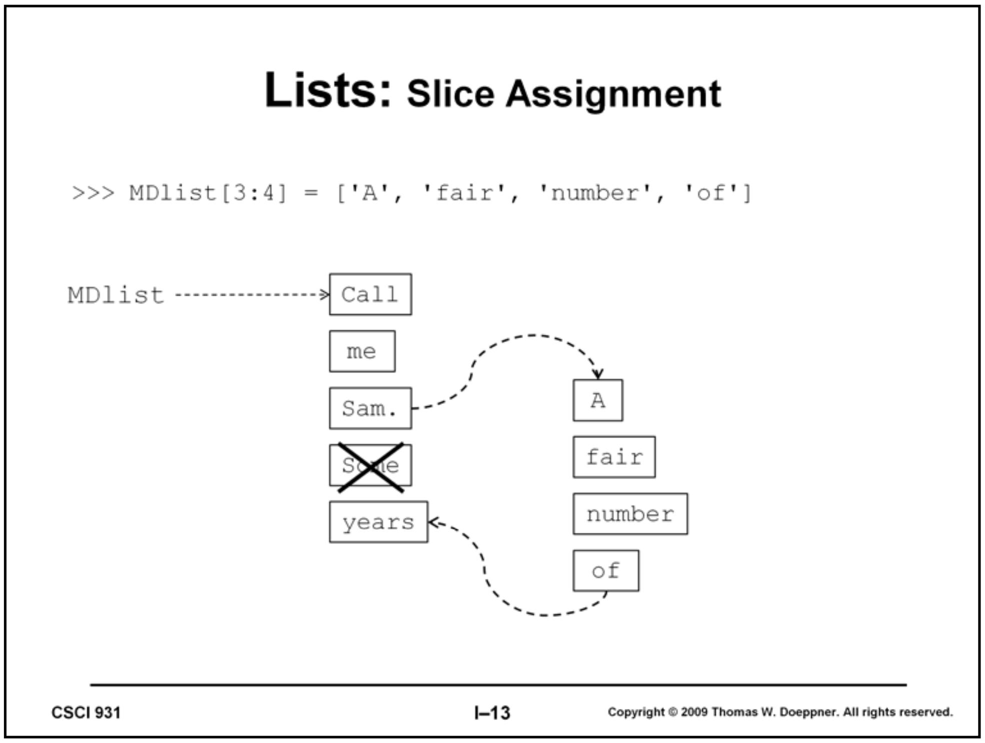
Lists: Item assignment

```
>>> MDlist[2] = 'Sam.'
```



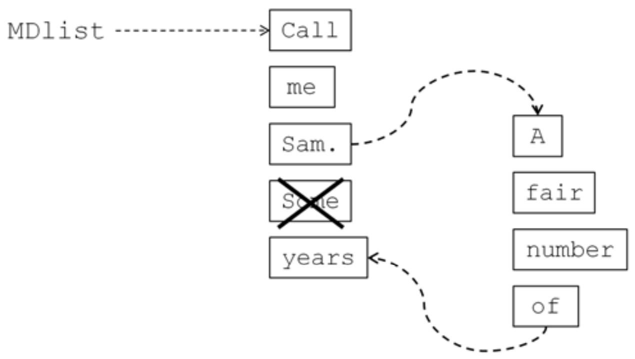
CSCI 931 I–12

The components of a list can be changed, as shown in the slide.

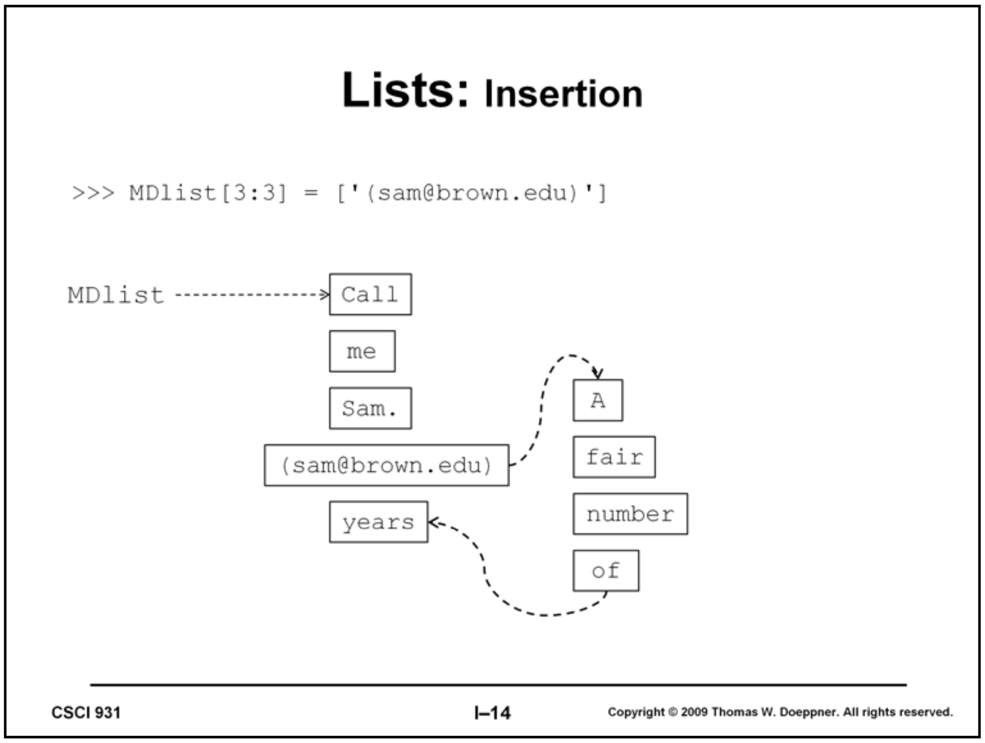


Lists: Slice Assignment

```
>>> MDlist[3:4] = ['A', 'fair', 'number', 'of']
```

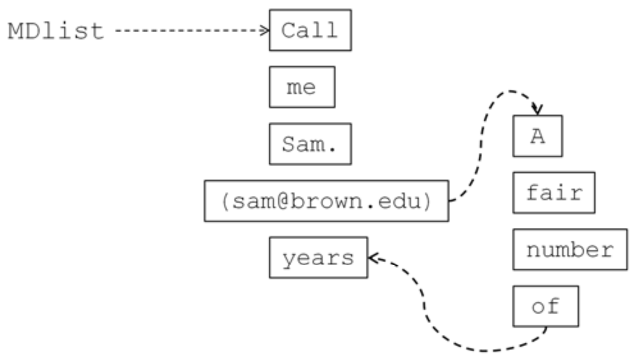


We can also replace a portion of a list with another list.



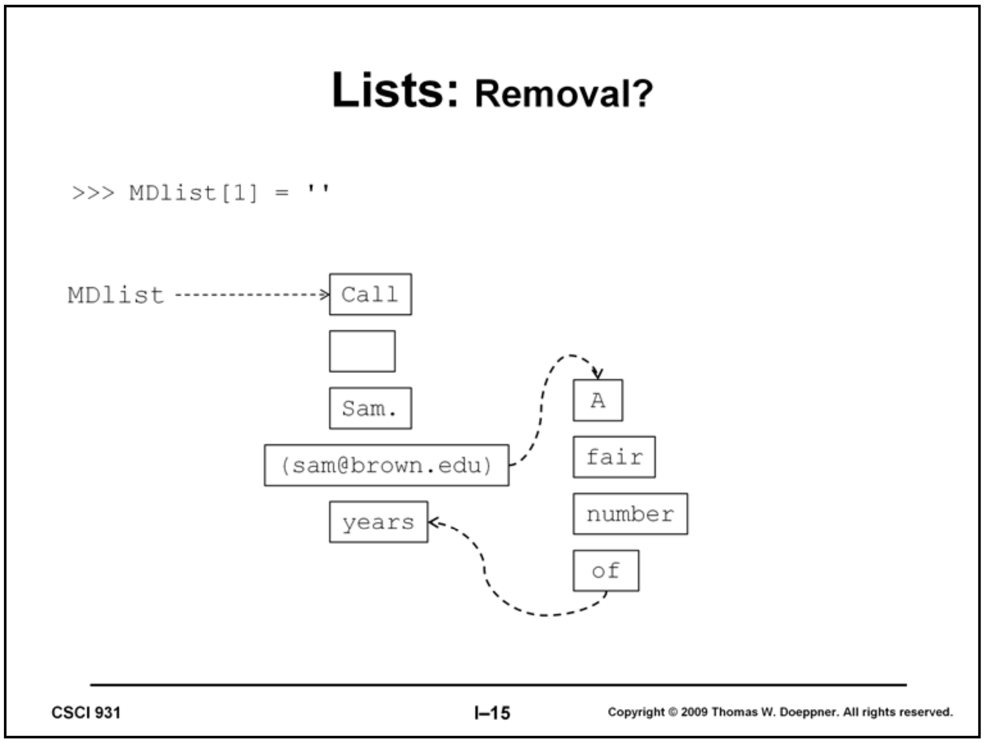
Lists: Insertion

```
>>> MDlist[3:3] = ['(sam@brown.edu)']
```



CSCI 931 I–14 Copyright © 2009 Thomas W. Doeppner. All rights reserved.

And we can insert a new item into a list.

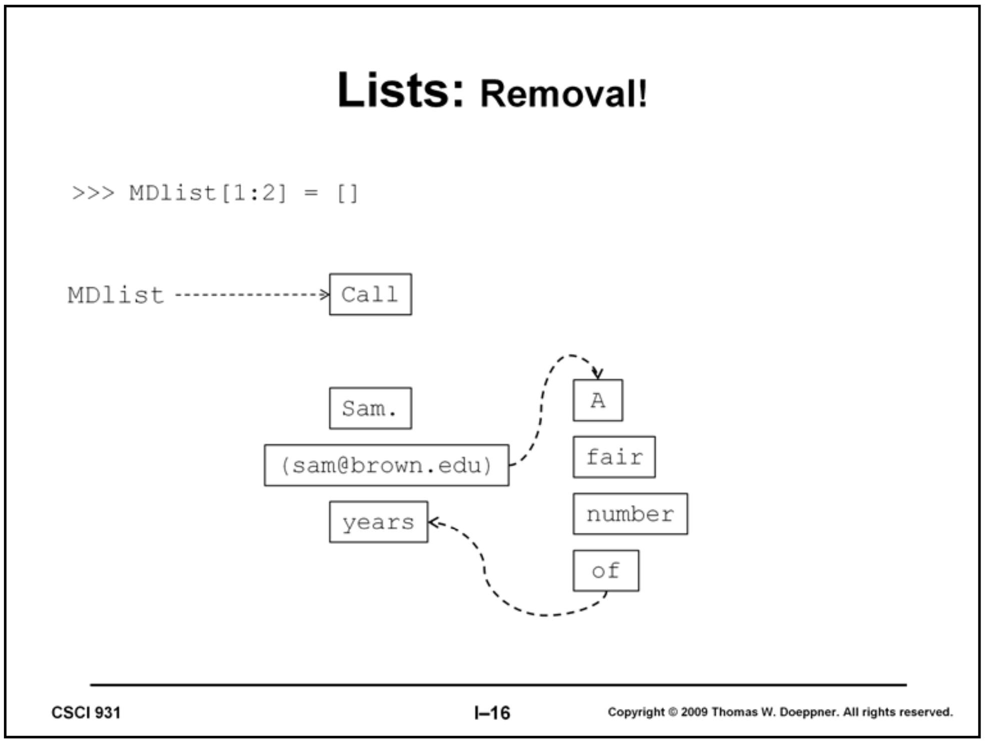


Lists: Removal?

```
>>> MDlist[1] = ''
```

CSCI 931 I–15 Copyright © 2009 Thomas W. Doeppner. All rights reserved.

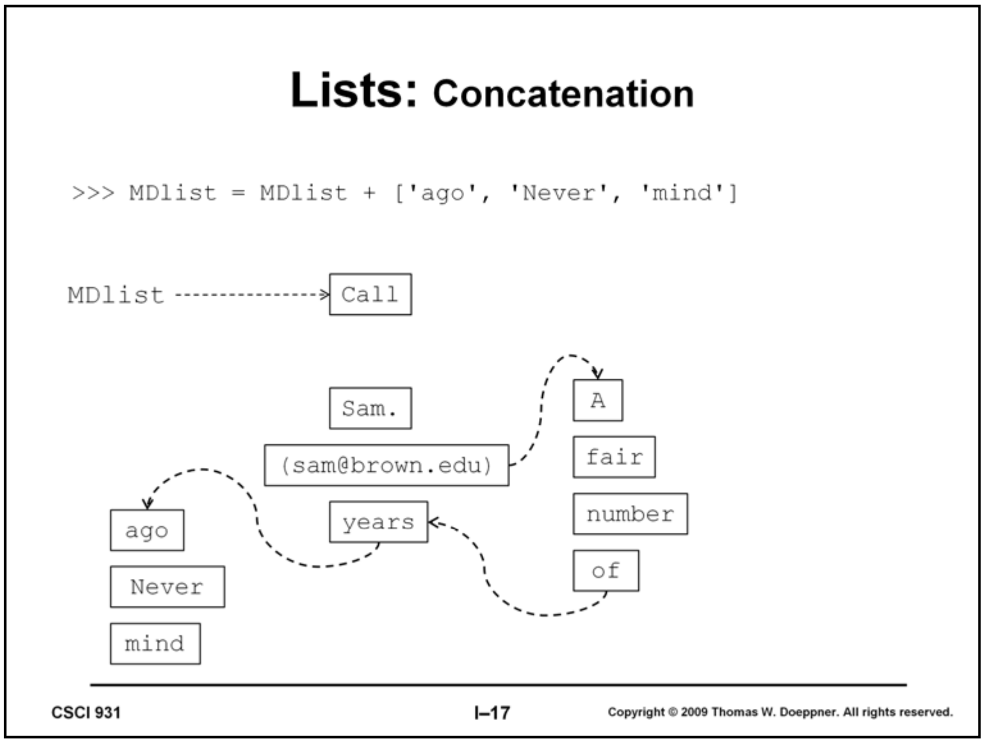
Removing an item from a list is somewhat tricky. What's done in the slide simply replaces an item of the list with the empty string (the string whose length is zero). This is actually a legitimate string, and thus the number of items in the list is unchanged.



Lists: Removal!

```
>>> MDlist[1:2] = []
```

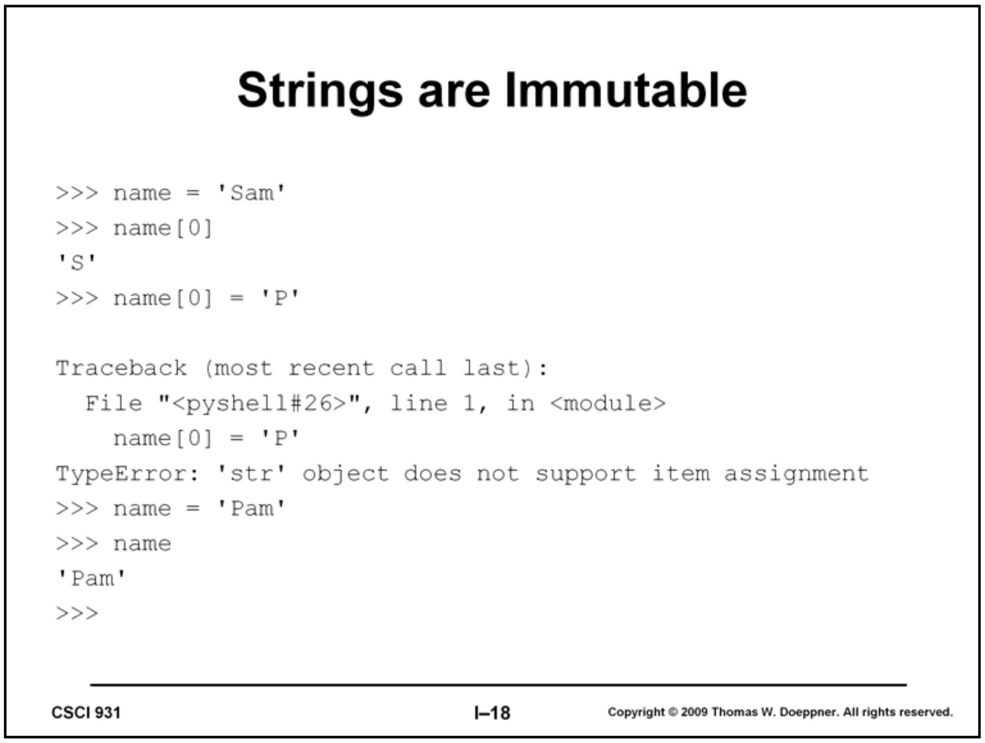
To really remove an item from a list, we have to replace it with the empty list.



Lists: Concatenation

```
>>> MDlist = MDlist + ['ago', 'Never', 'mind']
```

The plus operator, applied to lists, gives us list concatenation.

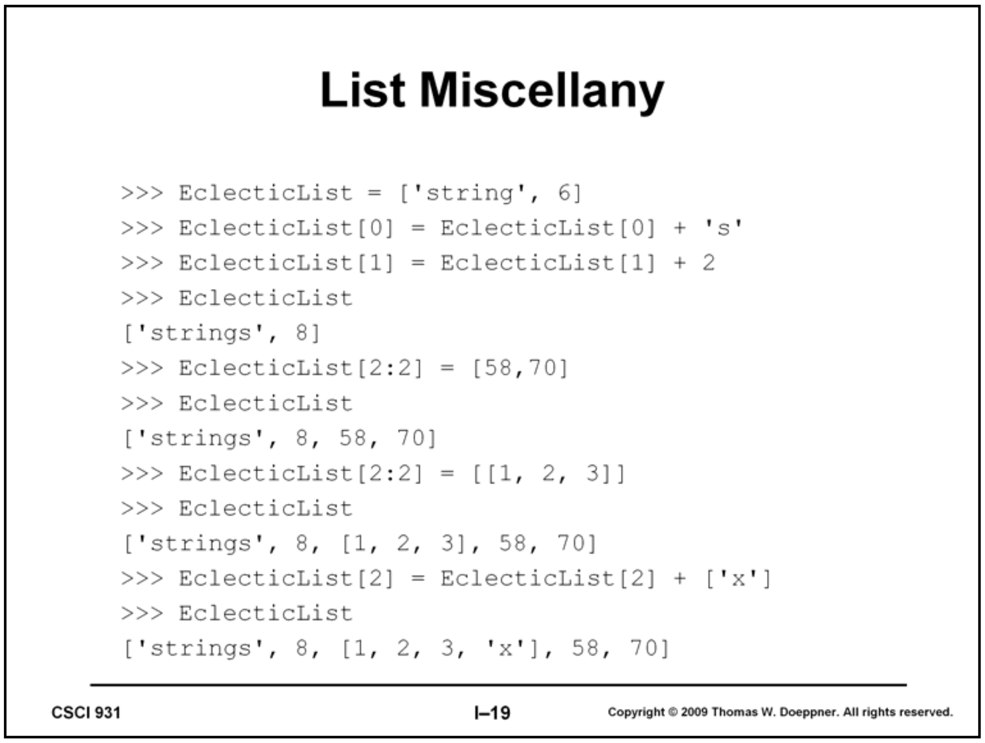


Strings are Immutable

```
>>> name = 'Sam'
>>> name[0]
'S'
>>> name[0] = 'P'

Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    name[0] = 'P'
TypeError: 'str' object does not support item assignment
>>> name = 'Pam'
>>> name
'Pam'
>>>
```

CSCI 931 I–18

We've seen how to modify a list, but note that Python will not let us modify a string. It considers strings to be *immutable*. Of course, we can cause an identifier to refer to a different string — this doesn't change the original string.



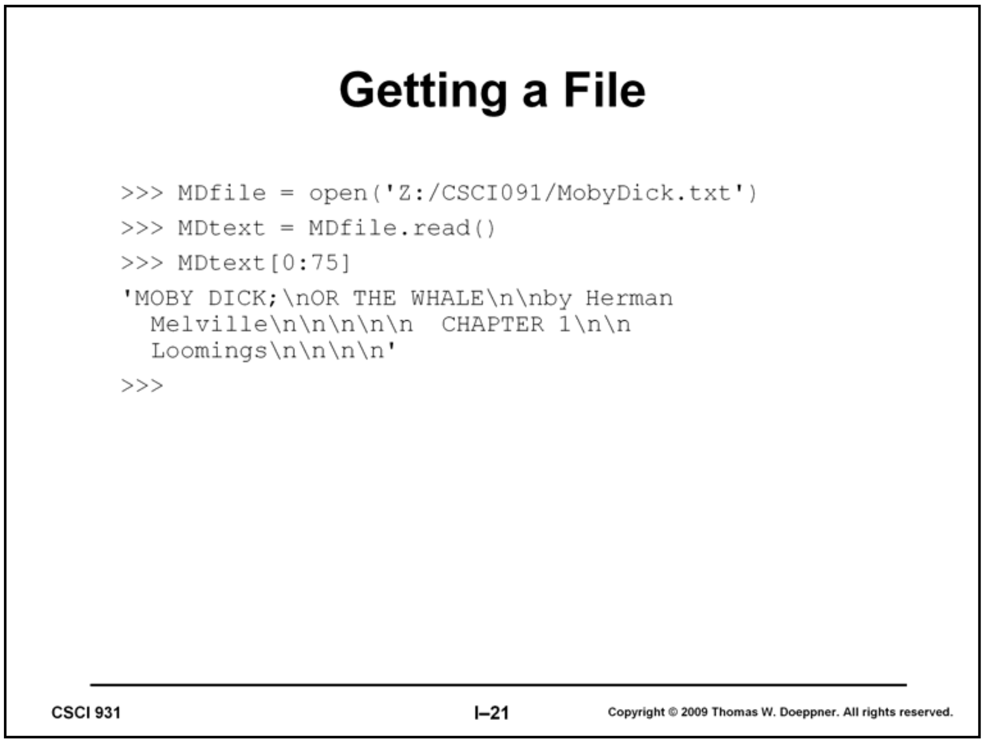
List Miscellany

```
>>> EclecticList = ['string', 6]
>>> EclecticList[0] = EclecticList[0] + 's'
>>> EclecticList[1] = EclecticList[1] + 2
>>> EclecticList
['strings', 8]
>>> EclecticList[2:2] = [58,70]
>>> EclecticList
['strings', 8, 58, 70]
>>> EclecticList[2:2] = [[1, 2, 3]]
>>> EclecticList
['strings', 8, [1, 2, 3], 58, 70]
>>> EclecticList[2] = EclecticList[2] + ['x']
>>> EclecticList
['strings', 8, [1, 2, 3, 'x'], 58, 70]
```

CSCI 931 I–19

Lists can be built from any kind of object, including other lists.

Lab: Fun with Python



Getting a File

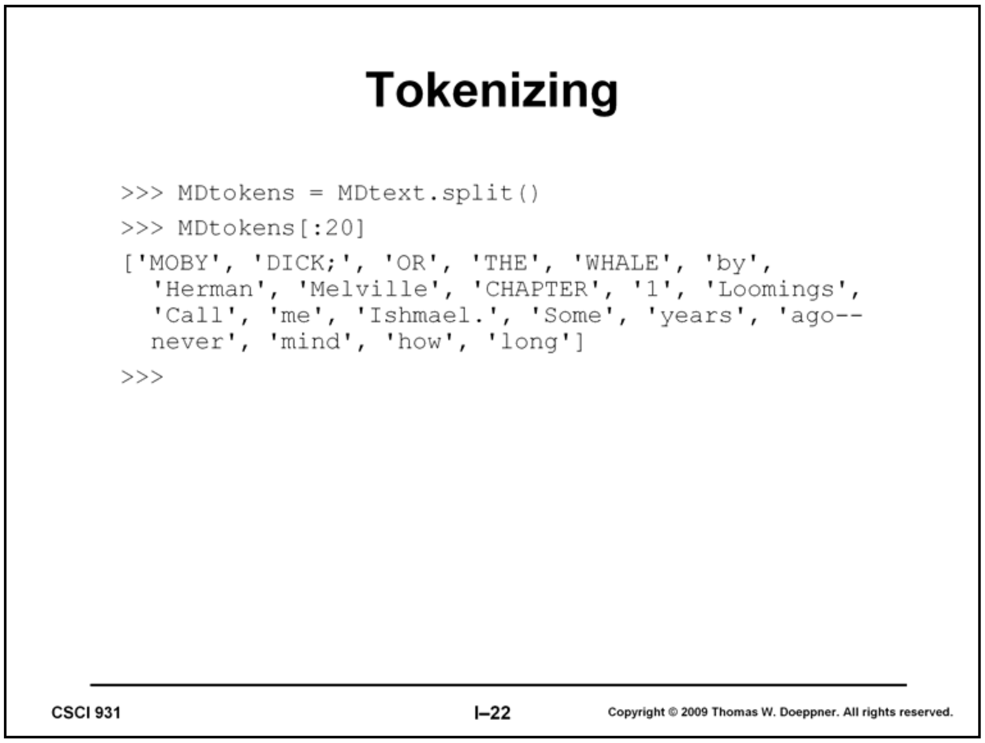
```
>>> MDfile = open('Z:/CSCI091/MobyDick.txt')
>>> MDtext = MDfile.read()
>>> MDtext[0:75]
'MOBY DICK;\nOR THE WHALE\n\nby Herman
  Melville\n\n\n\n\n  CHAPTER 1\n\n
  Loomings\n\n\n\n'
>>>
```

CSCI 931 I–21 Copyright © 2009 Thomas W. Doeppner. All rights reserved.

Let's now start working with some more reasonable text. First we have to get the text. So, let's assume we gone to project Gutenberg and have downloaded Moby Dick into a text file. We've also used WordPad to remove the copyright notices, etc. Before we can read the contents of the file into Python, we have to tell it what file we're interested. This is done via the function *open*, which asks the operating system to find the file for us and to make sure that we're allowed to read it. The call to open produces a value, provided by the operating system, that we are to use when referring to the file. We're not going to get into what this value actually is; we simply set MDfile to refer to it.

Next we see a new wrinkle in Python. The value referred to by MDfile is actually somewhat complex; among other things, we can use it to refer to a function that is used to read the contents of the file. This function is rather conveniently called *read*; the expression *MDfile.read()* produces a string that's the contents of the file. We set MDtext to refer to this (rather long) string.

Note that the pathname of the file that is passed to open uses forward slashes rather than reverse slashes, which would be the usual thing to do in Windows. Using forward slashes avoids potential problems that might come up if Python interprets them as "escape" characters that, combined with the next character, have a special meaning.


Tokenizing

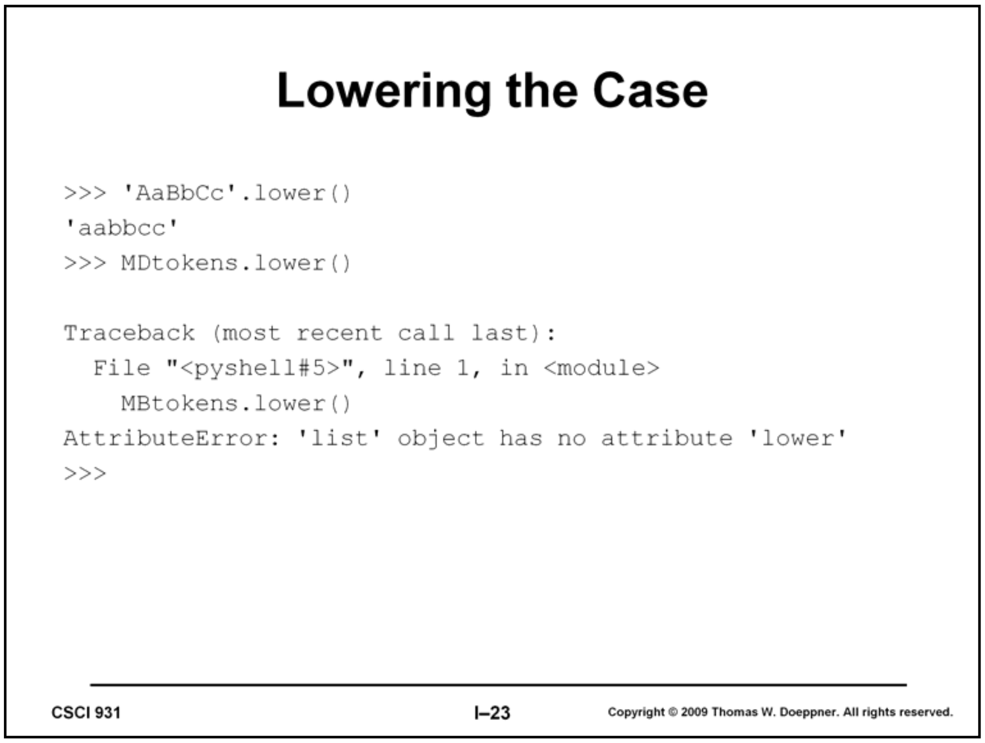
```
>>> MDtokens = MDtext.split()
>>> MDtokens[:20]
['MOBY', 'DICK;', 'OR', 'THE', 'WHALE', 'by',
  'Herman', 'Melville', 'CHAPTER', '1', 'Loomings',
  'Call', 'me', 'Ishmael.', 'Some', 'years', 'ago--
  never', 'mind', 'how', 'long']
>>>
```

CSCI 931 I–22 Copyright © 2009 Thomas W. Doeppner. All rights reserved.

If our goal is to work with the words of Moby Dick, then we'd like to convert the string into a list of words. Doing this, as we've already discussed, is rather complicated. Rather than doing it perfectly right now, we're going to do something that close to what we want, but not exactly what we want. (We'll do better later.)

"split" is a function that can be used only on strings. What it does is to break up the string into words, where words are delineated by "white space" (blanks, end of lines, and tabs), and produce a list containing those words. However, it doesn't know about punctuation, so our words contain punctuation such as periods, commas, etc.

You might think that we should invoke *split* by typing *split(MDtext)*. This, however, is not how it's done. The *split* we just described makes sense only when applied to strings. In principle we might define other splits, perhaps one that works with integers and another that works with lists. To make it clear that we're using the split that works with strings, we type *MDtext.split()*. What Python does with this is first to figure out what sort of thing *MDtext* refer to. In this case it refers to a string. Thus it uses the version of split that deals with strings and applies it to *MDtext*.

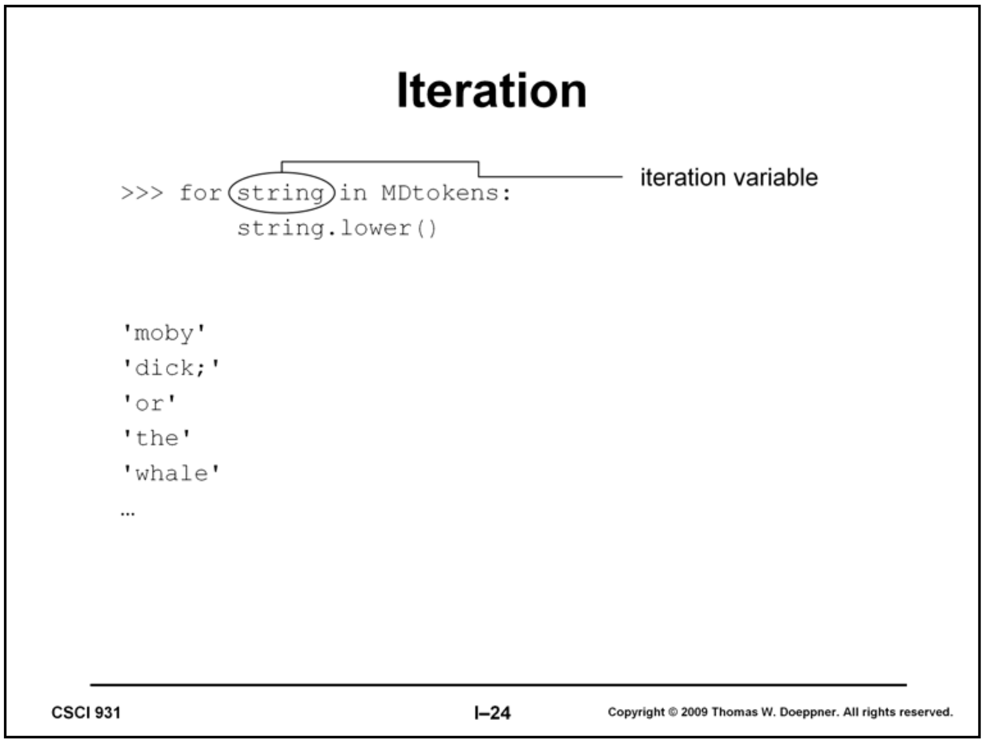


Lowering the Case

```
>>> 'AaBbCc'.lower()
'aabbcc'
>>> MDtokens.lower()

Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    MBtokens.lower()
AttributeError: 'list' object has no attribute 'lower'
>>>
```

CSCI 931 I–23

Here we have another example of a function that works only with strings. We actually try to use it on a list (MDtokens), but Python, in its own way, tells us that there is no version of *lower* that works on lists.



Iteration

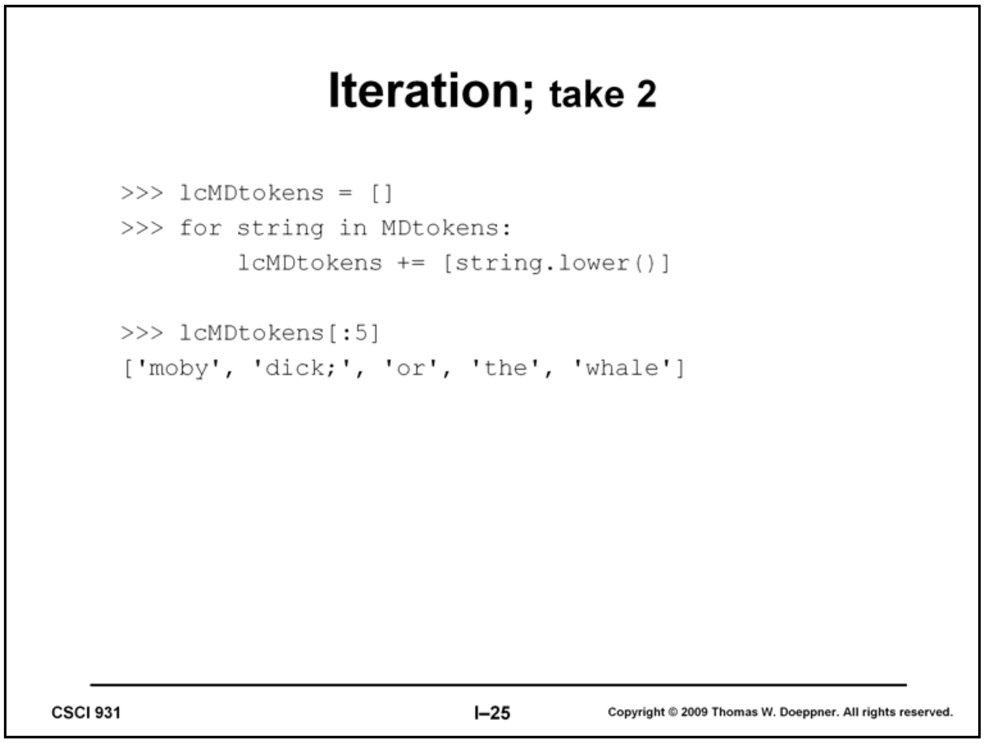
```
>>> for string in MDtokens:
        string.lower()


'moby'
'dick;'
'or'
'the'
'whale'
…
```

iteration variable
CSCI 931
I–24
Copyright © 2009 Thomas W. Doeppner. All rights reserved.

But we'd really like to use *lower* on each item in the list. For this we introduce the notion of *iteration*. This is something like the fill operation in Excel: you do something in one cell, then you tell Excel, say, to fill across, doing the same operation to a number of cells. Iteration gives us a means for, in this case, doing the same operation to all the items in a list.

This iteration statement starts with "for". Then comes the iteration variable, in this case "string". Following this is the keyword "in", followed by a reference to a list. Then comes a colon, indicating that more is to come on subsequent lines. The subsequent lines are executed a number of times; each time the iteration variable (string) is set to the next item in the list. Note that the second line is indented four spaces. IDLE does this for us automatically. To indicate to IDLE that you've got no more to type, simply type an empty line (i.e., hit "enter").

However, don't do exactly what's on the slide! Recall that our list is pretty long, so you'd end up watching the entire contents of Moby Dick appear on your screen, one word per line, somewhat slowly (why it's slow is due to problems with how Python itself is implemented).



Iteration; take 2

```
>>> lcMDtokens = []
>>> for string in MDtokens:
        lcMDtokens += [string.lower()]

>>> lcMDtokens[:5]
['moby', 'dick;', 'or', 'the', 'whale']
```

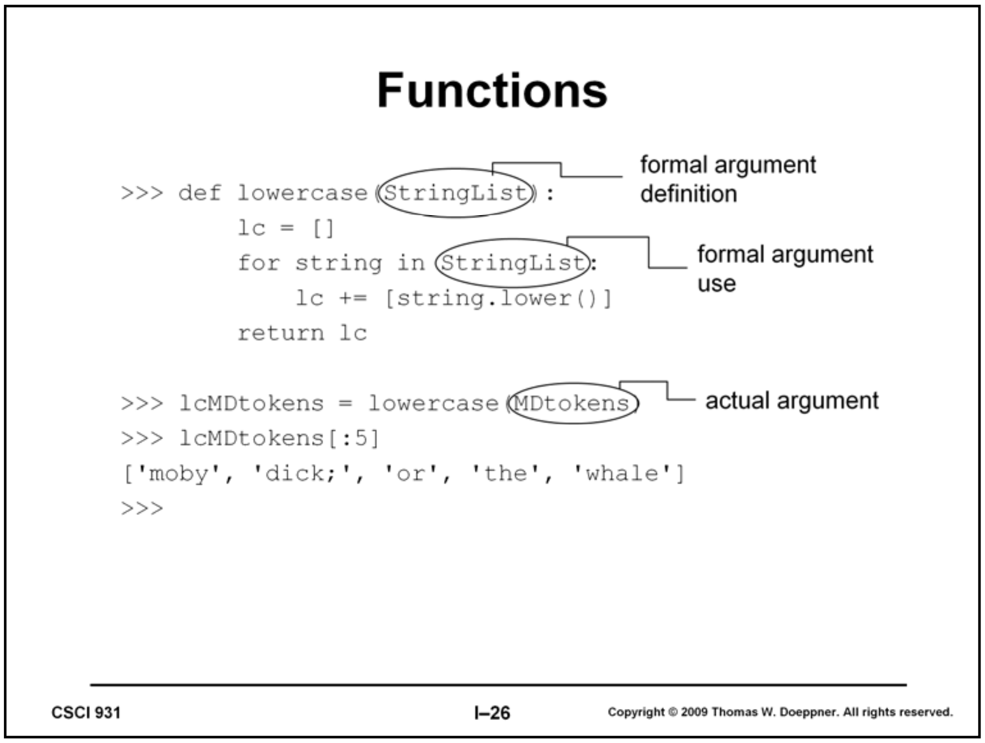
CSCI 931 I–25 Copyright © 2009 Thomas W. Doeppner. All rights reserved.

Here we do the same thing we were doing in the previous slide, but we put the result into a list, one word at a time.

Note that we must first initialize lcMDtokens to the empty list, just to make it clear to Python that it's supposed to refer to a list. This makes it clear when we use the "+" operator that what we have in mind is list concatenation.

Also note that we've introduced some shorthand. "a += b" is an easy way to type something that means the same as "a = a + b".

Further note that the expression "string.lower()" is placed within square brackets. This is important because string.lower() produces a string, but we are using the "+" operator to concatenate two lists. Thus we must turn the string into a (single-item) list.



Functions

```
>>> def lowercase(StringList):
        lc = []
        for string in StringList:
            lc += [string.lower()]
        return lc

>>> lcMDtokens = lowercase(MDtokens)
>>> lcMDtokens[:5]
['moby', 'dick;', 'or', 'the', 'whale']
>>>
```

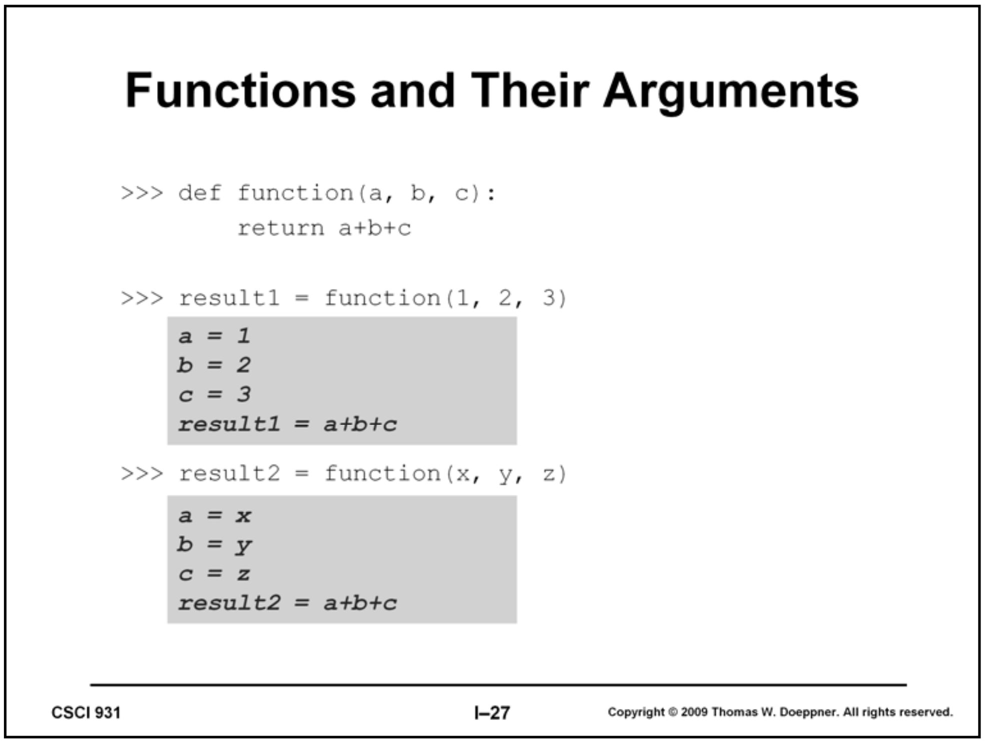
formal argument definition
formal argument use
actual argument

CSCI 931 I–26 Copyright © 2009 Thomas W. Doeppner. All rights reserved.

Rather than having to type in the contents of the previous slide every time we want to perform the lower operation on each item in a list, we can produce our own function that does the same thing. We start a function definition in Python with the word "def" and follow it with the name of the new function. Then, in parentheses, is a list of zero or more arguments. (These are known as the *formal arguments* to the function.) Then comes a colon, indicating that we're not done yet. On the next lines come the body of the function, which spell out what the function actually does. The function produces a value that is the value of whatever expression follows *return*.

When we invoke the function we provide *actual arguments*. The formal arguments are set to refer to the corresponding actual arguments. Thus, in the example of the slide, we invoke *lowercase*, providing it the actual argument *MDtokens*. The body of lowercase is executed, after setting the formal argument *StringList* to refer to *MDtokens*.

Note that the body of the function is indented four spaces. The body of the "for" statement is indented a further four spaces.

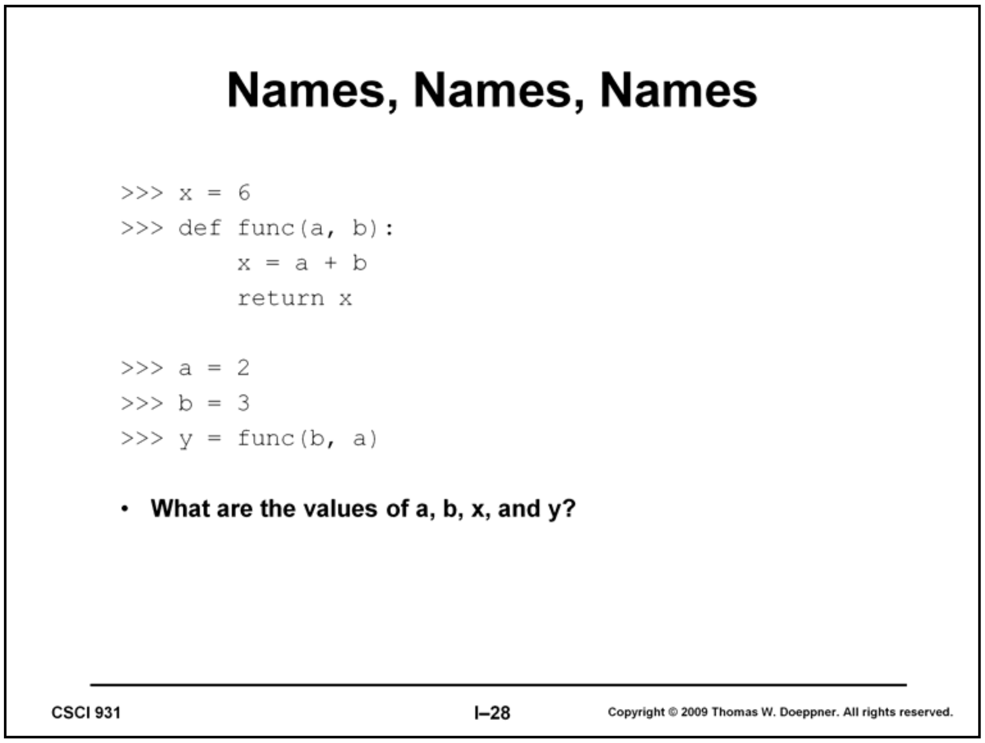


Functions and Their Arguments

```
>>> def function(a, b, c):
        return a+b+c

>>> result1 = function(1, 2, 3)
    a = 1
    b = 2
    c = 3
    result1 = a+b+c
>>> result2 = function(x, y, z)
    a = x
    b = y
    c = z
    result2 = a+b+c
```

CSCI 931 I–27 Copyright © 2009 Thomas W. Doeppner. All rights reserved.

Here we see a simpler function, called *function*. It has three formal arguments; we see within the boxes the effect of each invocation.



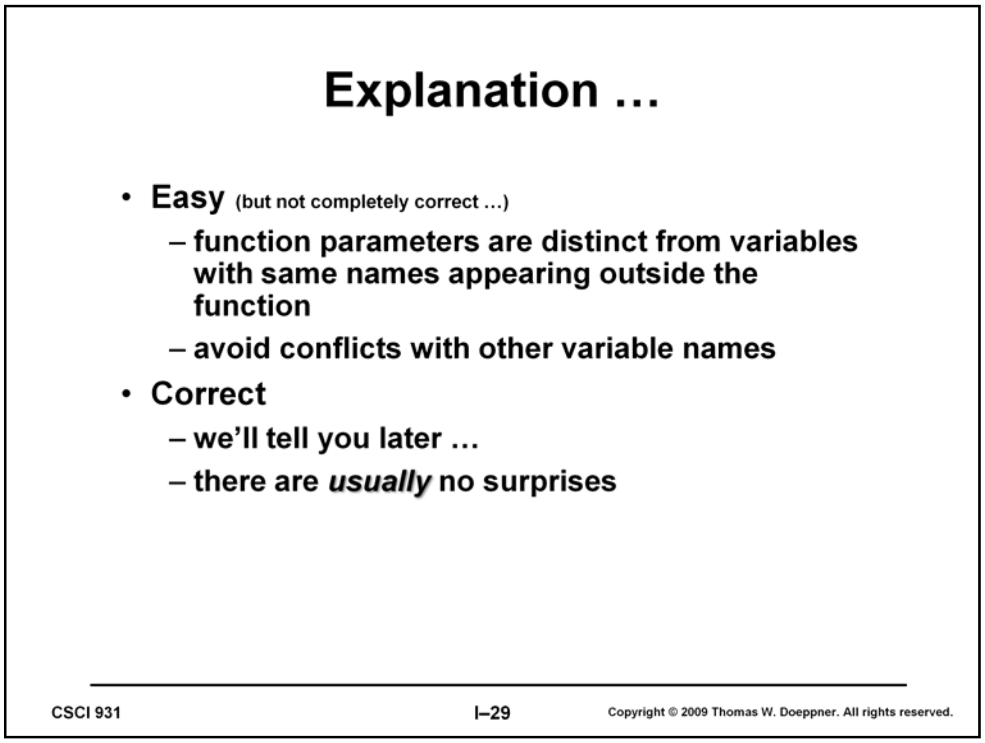
Names, Names, Names

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

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

- **What are the values of a, b, x, and y?**

We've got to be pretty careful with our use of names, since there can be some apparent conflicts.

(Run this code yourself if you're unsure of the answer to the question.)

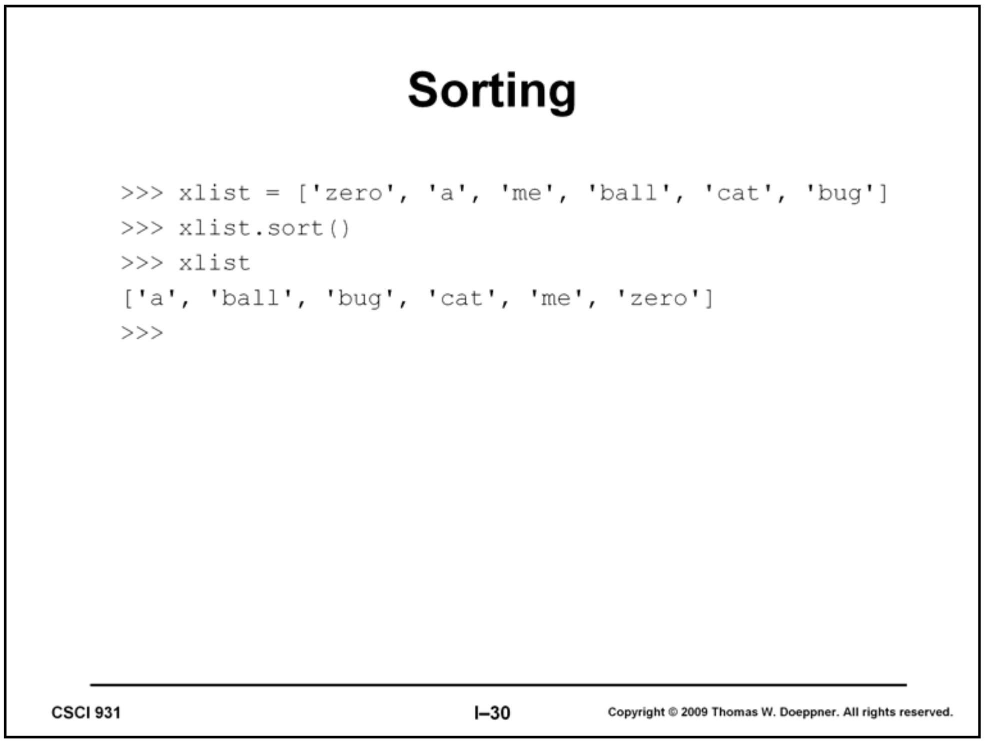


Explanation …

- **Easy** (but not completely correct …)
  - function parameters are distinct from variables with same names appearing outside the function
  - avoid conflicts with other variable names
- **Correct**
  - we'll tell you later …
  - there are ***usually*** no surprises

CSCI 931 I–29

There are well defined rules that explain what seemingly conflicting names actually refer to. Though the rules aren't terribly complicated, we'll explain them a bit later on.

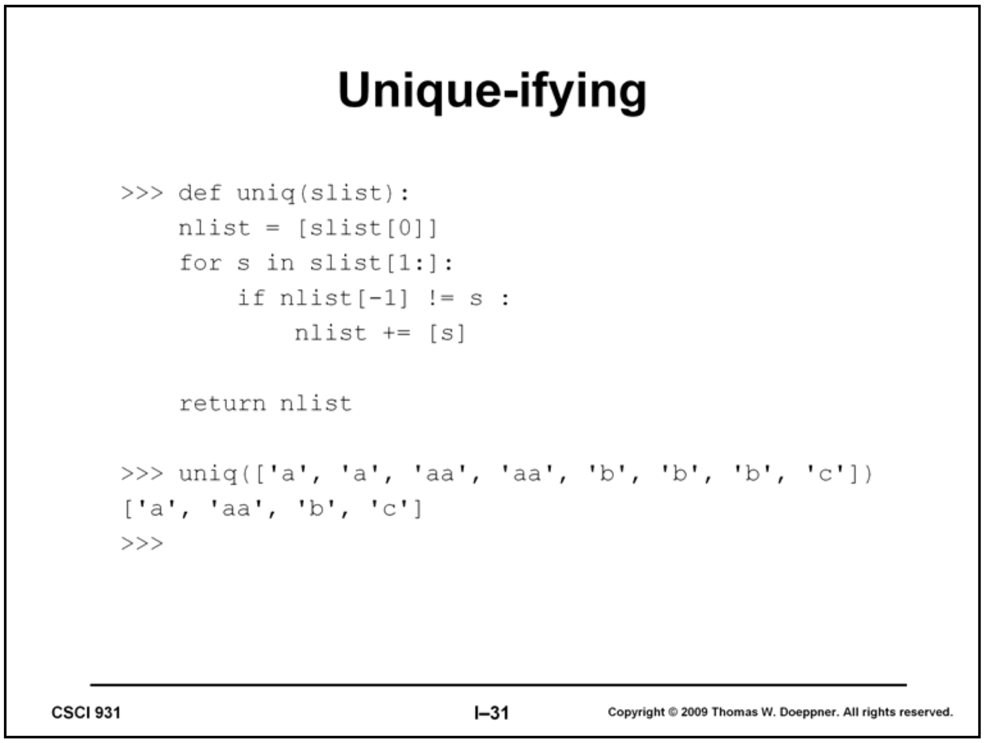


Sorting

```
>>> xlist = ['zero', 'a', 'me', 'ball', 'cat', 'bug']
>>> xlist.sort()
>>> xlist
['a', 'ball', 'bug', 'cat', 'me', 'zero']
>>>
```

CSCI 931 I–30

Another list function is *sort*. Unlike some of the other functions we've seen, sort doesn't produce a new list, but sorts the contents of the given list. (This is know as *sorting in place.)*



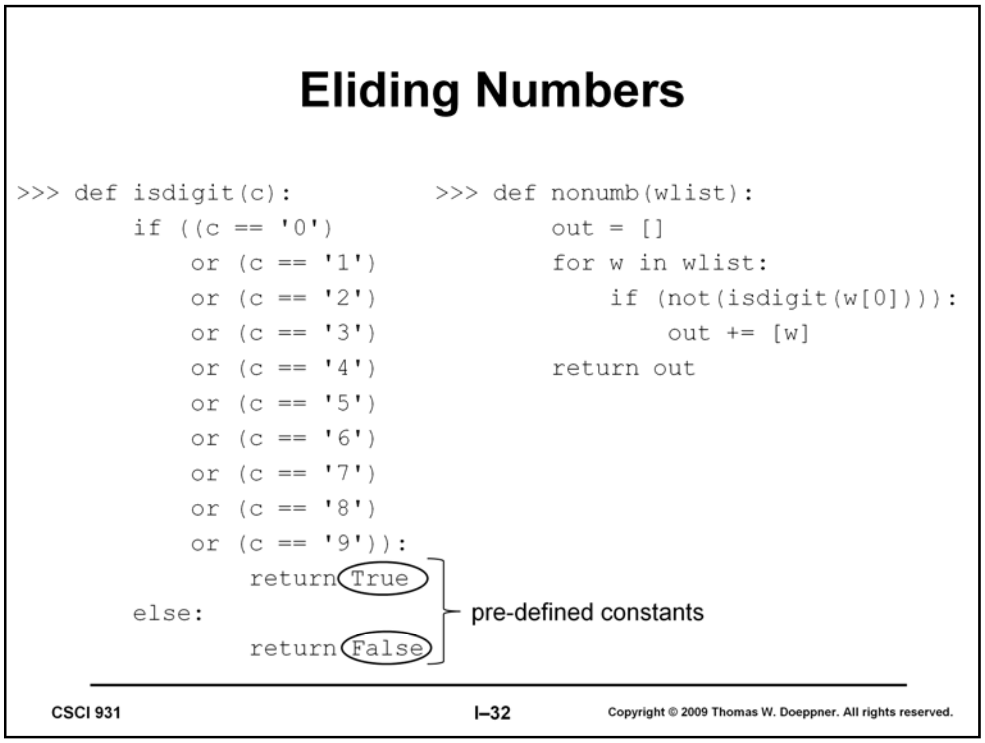
Unique-ifying

```
>>> def uniq(slist):
    nlist = [slist[0]]
    for s in slist[1:]:
        if nlist[-1] != s :
            nlist += [s]

    return nlist

>>> uniq(['a', 'a', 'aa', 'aa', 'b', 'b', 'b', 'c'])
['a', 'aa', 'b', 'c']
>>>
```

CSCI 931 I–31

Here's a program to remove adjacent repeated items in a list. This function makes use of *conditional* statements, otherwise known as *if* statements. In the form shown here, the expression after the "if" is evaluated. If it's true, then the statement after the "if" is executed, otherwise it is not. "!=" means "not equal" — thus the next statement is executed if the nlist[-1] is not equal to s.

Note that we've introduced the use of -1 as a list index. It refers to the last element of the list. (An index of -2 refers to the next-to-last element of a list, and so forth.)



Eliding Numbers

```
>>> def isdigit(c):
        if ((c == '0')
            or (c == '1')
            or (c == '2')
            or (c == '3')
            or (c == '4')
            or (c == '5')
            or (c == '6')
            or (c == '7')
            or (c == '8')
            or (c == '9')):
                return True
        else:
                return False
```

```
>>> def nonumb(wlist):
        out = []
        for w in wlist:
            if (not(isdigit(w[0]))):
                out += [w]
        return out
```

CSCI 931 I–32 Copyright © 2009 Thomas W. Doeppner. All rights reserved.

Here we define a function to test if a character is a digit. We'll see more succinct ways of doing this later, but this is pretty straightfoward. Note that we've extended the if statement with an "else" clause. If the expression following the "if" turns out to be false, then the statement following the "else" is executed. We've also used "or" to string together a sequence of expressions. If anyone of them is true, then the entire combined expression is true.

The *isdigit* function is used a building block for the *nonumb* function, which produces a new list from its argument that has all words starting with a digit removed. Note that *True* and *False* are constants, pre-defined by Python.