

CSc 120

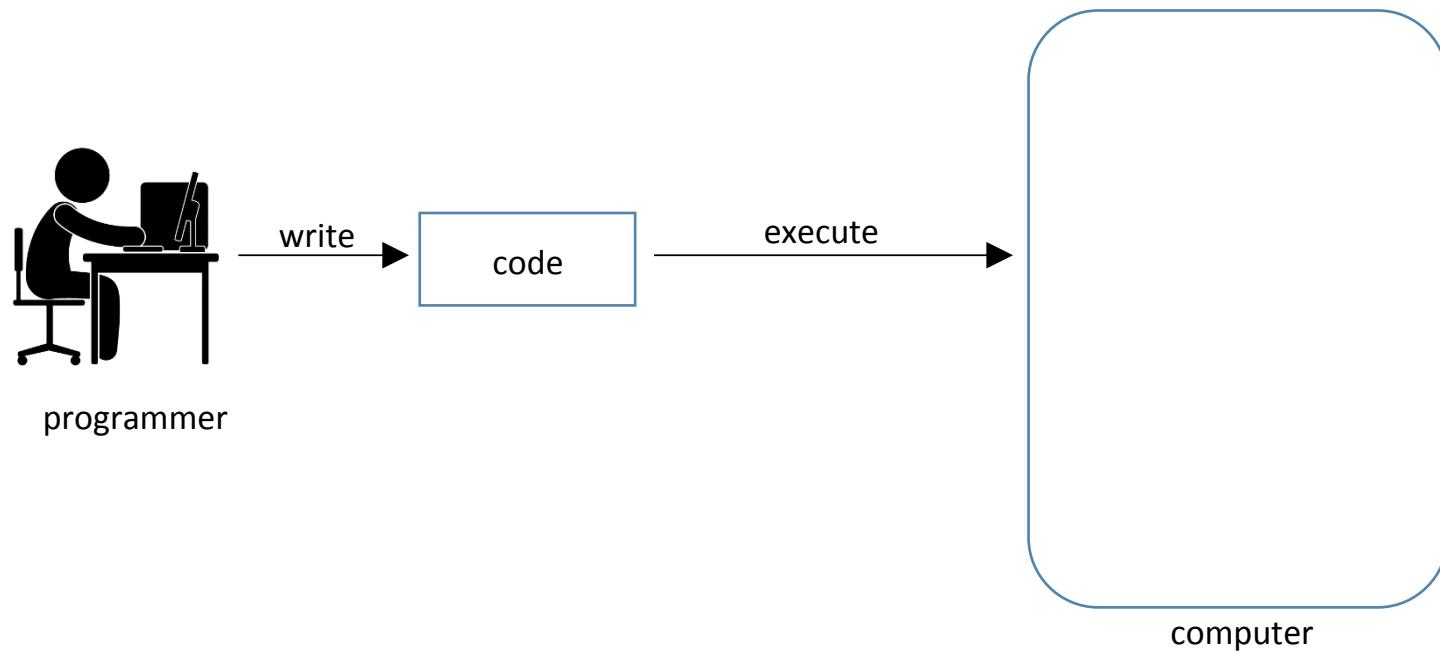
Introduction to Computer Programming II

*Adapted from slides by
Dr. Saumya Debray*

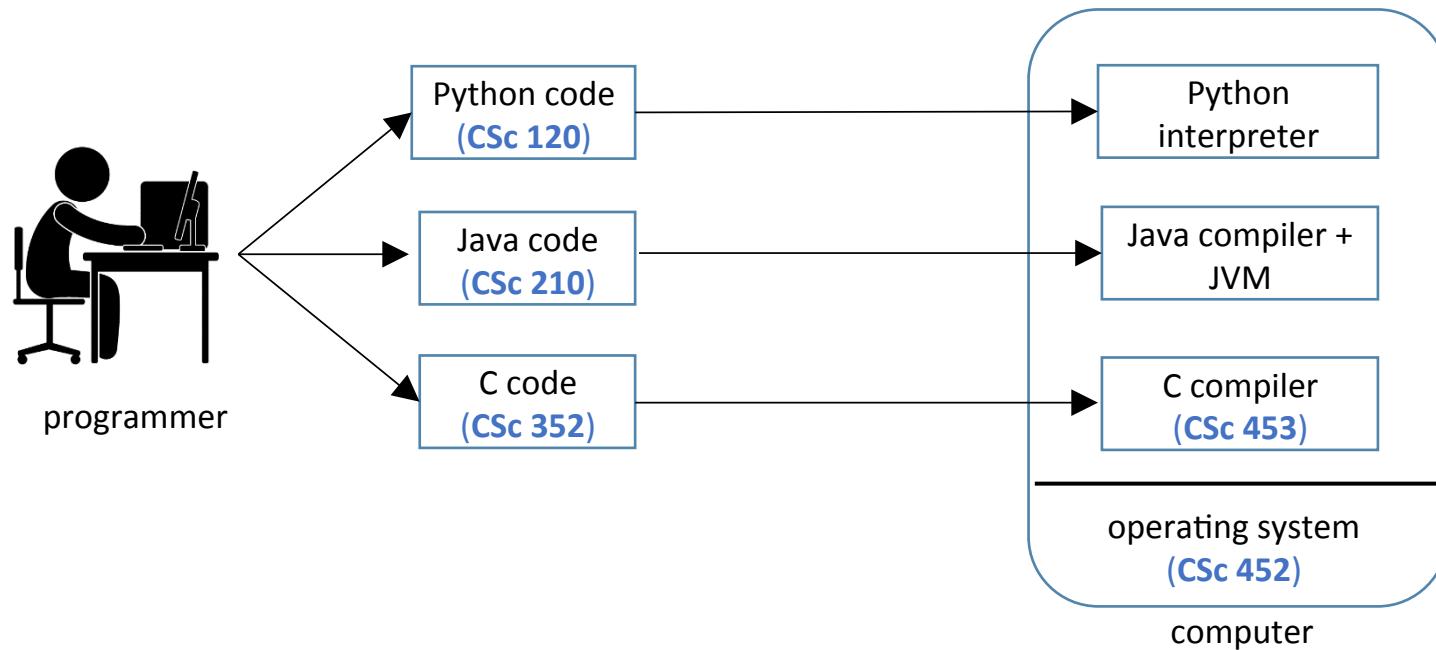
01: Python review

this class in context

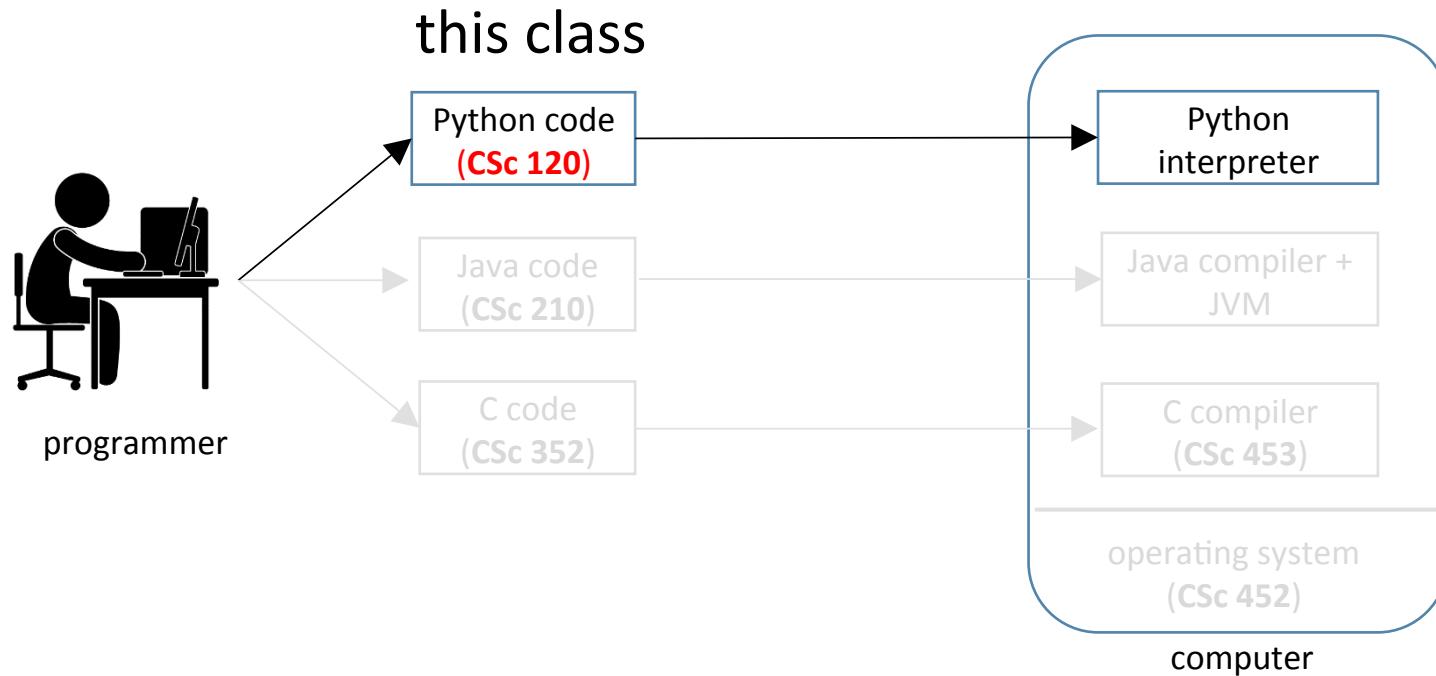
Computer programming



Computer programming



Computer programming



getting started

Python language and environment

- Language: Python 3
 - free download from
<https://www.python.org/downloads>
 - documentation available at
<https://www.python.org/doc>
 - tutorial
 - beginner's guide
 - language reference
 - setup and usage, HOWTOs, FAQs

Python language and environment

- Programming environment: idle (or idle3)
 - comes bundled with Python download
 - provides:
 - interactive Python shell
 - debugger
 - execution from a file

Surprises if coming from C, C++, Java

- No variable declarations
- Indentation instead of { }
- Flexible `for` loop
- Built-in data structures (lists, dictionaries, tuples, sets)
- Arbitrary-precision integers
- Devision differences
- Garbage collection (also in Java)
 - no explicit allocation/deallocation

python review: variables, expressions, assignment

python basics

```
>>> x = 4
>>> y = 5
>>> z = x + y
>>> x
4
>>> y
5
>>> z
9
>>> y = z * 2
>>> y
18
>>>
```

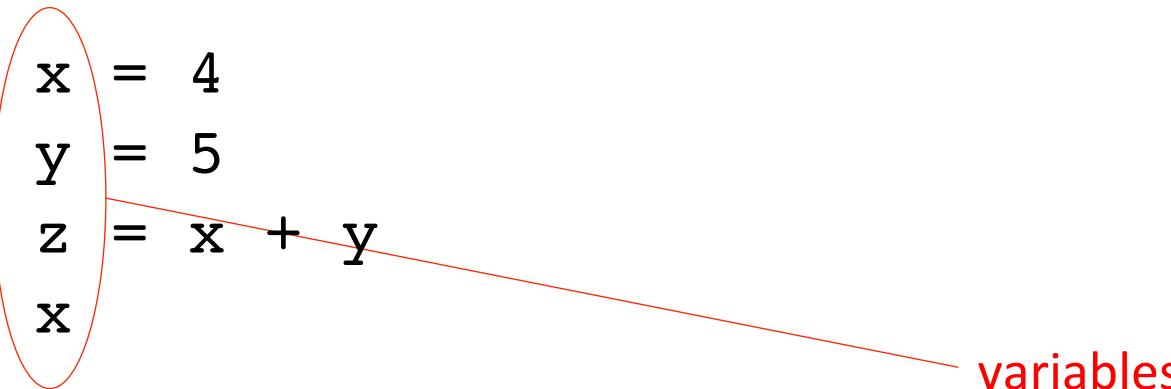
python basics

```
>>> x = 4
>>> y = 5
>>> z = x + y
>>> x
4
>>> y
5
>>> z
9
>>> y = z * 2
>>> y
18
>>>
```

>>> : python interpreter's prompt
black: user input (keyboard)
blue: python interpreter output

python basics

```
>>> x = 4
>>> y = 5
>>> z = x + y
>>> x
4
>>> y
5
>>> z
9
>>> y = z * 2
>>> y
18
>>>
```



variables

python basics

```
>>> x = 4
>>> y = 5
>>> z = x + y
>>> x
4
>>> y
5
>>> z
9
>>> y = z * 2
>>> y
18
>>>
```

expressions

python basics

```
>>> x = 4  
>>> y = 5  
>>> z = x + y
```

```
>>> x
```

4

```
>>> y
```

5

```
>>> z
```

9

```
>>> y = z * 2
```

```
>>> y
```

18

```
>>>
```

assignment
statements

python basics

```
>>> x = 4  
>>> y = 5  
>>> z = x + y  
  
4  
>>> y  
5  
>>> z  
9  
  
>>> y = z * 2  
>>> y  
18  
>>>
```

typing in an expression causes its value to be printed

python basics

```
>>> x = 4  
>>> y = 5  
>>> z = x + y  
>>> x  
4  
>>> y  
5  
>>> z  
9  
>>> y = z * 2  
>>> y  
18  
>>>
```

- variables:

- names begin with letter or '_'
- don't have to be declared in advance
 - type determined at runtime

- expressions:

- all the usual arithmetic operators

Multiple (aka parallel) assignment

```
>>>  
>>> x, y, z = 11, 22, 33  
>>> x  
11  
>>> y  
22  
>>> z  
33  
>>>
```

Assigns to multiple variables at the same time

$x_1, x_2, \dots, x_n = exp_1, exp_2, \dots, exp_n$

Behavior:

1. exp_1, \dots, exp_n evaluated (L-to-R)
2. x_1, \dots, x_n are assigned (L-to-R)

EXERCISE

```
>>> x = 3
```

```
>>> y = 4
```

```
>>> z = (2*x - 1 == y+1)
```

```
>>> z
```

← *what value is printed out for z?*

EXERCISE

```
>>> x = 3
```

```
>>> y = 4
```

```
>>> sum, diff, prod = x + y, x - y, x * y
```

```
>>> prod+diff
```

← *what is the value printed out?*

python review: reading user input I: `input()`

Reading user input I: `input()`

```
>>> x = input()  
13579  
  
>>> x  
'13579'  
  
>>> y = input('Type some input: ')  
Type some input: 23  
  
>>> y  
'23'  
  
>>> z = input('More input: ')  
More input: 567  
  
>>> z  
'567'  
  
>>>
```

Reading user input I: `input()`

```
>>> x = input()
13579
>>> x
'13579'
>>> y = input('Type some input: ')
Type some input: 23
>>> y
'23'
>>> z = input('More input: ')
More input: 567
>>> z
'567'
>>>
```

input statement:

- reads input from the keyboard
- returns the value read
 - (a string)

Reading user input I: `input()`

```
>>> x = input()
```

```
13579
```

```
>>> x
```

```
'13579'
```

```
>>> y = input('Type some input: ')
```

```
Type some input: 23
```

```
>>> y
```

```
'23'
```

```
>>> z = input('More input: ')
```

```
More input: 567
```

```
>>> z
```

```
'567'
```

```
>>>
```

input statement:

- reads input from the keyboard
- returns the value read
 - (a string)
- takes an optional argument
 - if provided, serves as a prompt

Reading user input I: `input()`

```
>>>  
>>> x = input()  
12  
>>> x  
'12'  
>>> y = x / 2
```

the value read in is represented as a string

- string \equiv sequence of characters

Traceback (most recent call last):

```
  File "<pyshell#59>", line 1, in <module>  
    y = x / 2
```

`TypeError: unsupported operand type(s) for /: 'str' and 'int'`

```
>>>
```

Reading user input I: `input()`

```
>>>  
>>> x = input()  
12  
>>> x  
'12'  
>>> y = x / 2
```

the value read in is represented as a string

- string \equiv sequence of characters

~~Traceback (most recent call last):
File "<pyshell#59>", line 1, in <module>
y = x / 2
TypeError: unsupported operand type(s) for /: 'str' and 'int'~~

```
>>>
```

- `TypeError`: indicate an error due to wrong type

Reading user input I: `input()`

```
>>>
```

```
>>> x = input()
```

```
12
```

```
>>> x
```

```
'12'
```

```
>>> y = x / 2
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#59>", line 1, in <module>
```

```
    y = x / 2
```

```
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

```
>>> y = int(x) / 2
```

```
>>> y
```

```
6.0
```

```
>>>
```

the value read in is represented as a string

- string \equiv sequence of characters
- `TypeError`: indicates an error due to a wrong type

- Fix: explicit type conversion

python review: basics of strings

Basics of strings

```
>>> x = "abcd"  
>>> y = 'efgh'  
>>> z = "efgh"  
>>>
```

Basics of strings

```
>>> x = "abcd"  
>>> y = 'efgh'  
>>> z = "efgh"  
>>>
```

either single-quotes (at both ends)
or double-quotes (at both ends)

Basics of strings

```
>>> text = input('Enter a string: ')
```

```
Enter a string: abcdefghi
```

```
>>> text
```

```
'abcdefghijklm'
```

```
>>> text[0]
```

```
'a'
```

```
>>> text[1]
```

```
'b'
```

```
>>> text[27]
```

Traceback (most recent call last):

```
  File "<pyshell#153>", line 1, in <module>
```

```
    text[27]
```

IndexError: string index out of range

```
>>>
```

a string is a sequence (array) of characters

- we can index into a string to get the characters

Basics of strings

```
>>> text = input('Enter a string: ')
```

```
Enter a string: abcdefghi
```

```
>>> text
```

```
'abcdefghi'
```

```
>>> text[0]
```

```
'a'
```

```
>>> text[1]
```

```
'b'
```

```
>>> text[27]
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#153>", line 1, in <module>
```

```
    text[27]
```

```
IndexError: string index out of range
```

```
>>>
```

a string is a sequence (array) of characters

- we can index into a string to get the characters

indexing beyond the end of the string gives an **IndexError** error

Basics of strings

```
>>> text = input('Enter a string: ')
```

```
Enter a string: abcdefghi
```

```
>>> text
```

```
'abcdefghi'
```

```
>>> text[0]
```

```
'a'
```

```
>>> text[1]
```

```
'b'
```

```
>>> text[27]
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#153>", line 1, in <module>
```

```
    text[27]
```

```
IndexError: string index out of range
```

```
>>>
```

a string is a sequence (array) of characters

- we can index into a string to get the characters
- each character is returned as a string of length 1

Intuitively, a *character* is a single letter, digit, punctuation mark, etc.

E.g.: 'a'

'5'

'\$'

Basics of strings

```
>>> x = '0123456789'
```

```
>>>
```

```
>>> x[0]
```

'0'

```
>>> x[1]
```

'1'

```
>>> x[2]
```

'2'

```
>>>
```

```
>>> x[-1]
```

'9'

```
>>> x[-2]
```

'8'

```
>>> x[-3]
```

'7'

```
>>>
```

$x[i]$: if $i \geq 0$ (i.e., non-negative values):

- indexing is done from the beginning of the string
- the first letter has index 0

$x[i]$: if $i < 0$ (i.e., negative values):

- indexing is done from the end of the string
- the last letter has index -1

Basics of strings

```
>>> x = '0123456789'
```

```
>>>
```

```
>>> x[0]
```

```
'0'
```

```
>>> x[1]
```

```
'1'
```

```
>>> x[2]
```

```
'2'
```

```
>>>
```

```
>>> x[-1]
```

```
'9'
```

```
>>> x[-2]
```

```
'8'
```

```
>>> x[-3]
```

```
'7'
```

```
>>>
```

$x[i]$: if $i \geq 0$ (i.e., non-negative values):

- indexing is done from the beginning of the string
- the first letter has index 0

$x[i]$: if $i < 0$ (i.e., negative values):

- indexing is done from the end of the string
- the last letter has index -1

EXERCISE

```
>>> x = 'a'
```

```
>>> x == x[0]
```



what do you think will be printed here?

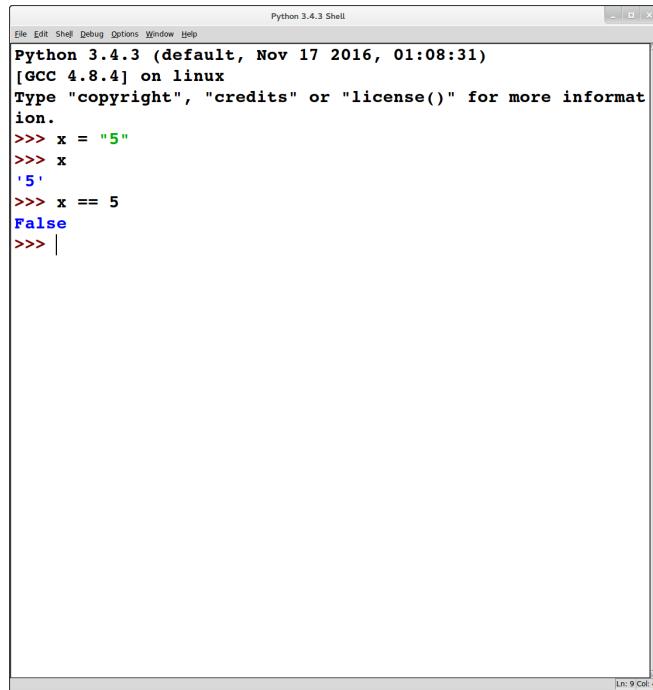
EXERCISE

```
>>> x = 'apple'
```

```
>>> x[2] == x[-2]
```

what do you think will be printed here?

Basics of strings



A screenshot of the Python 3.4.3 Shell window. The menu bar includes File, Edit, Shelf, Debug, Options, Window, and Help. The title bar says "Python 3.4.3 Shell". The code input area shows:

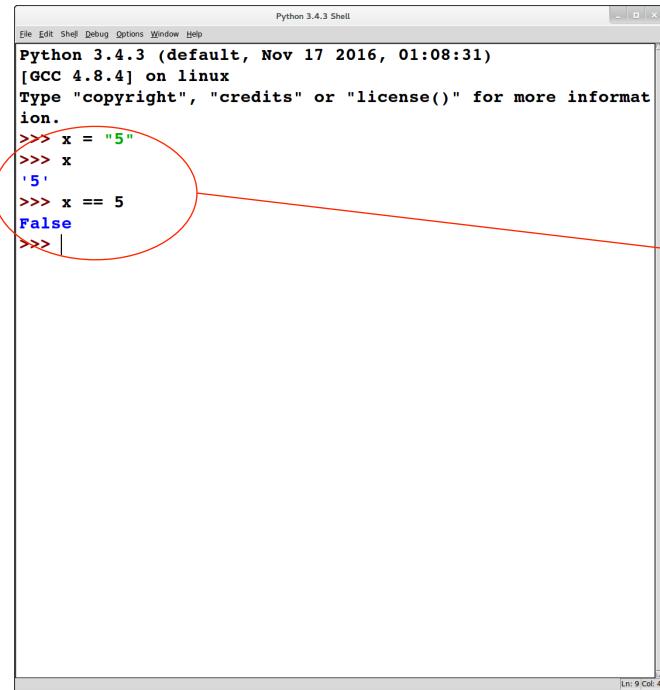
```
Python 3.4.3 (default, Nov 17 2016, 01:08:31)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> x = "5"
>>> x
'5'
>>> x == 5
False
>>> |
```

The status bar at the bottom right indicates "Ln: 9 Col: 4".

Inside a computer, a character is represented as a number (its "ASCII value")

Basics of strings



A screenshot of the Python 3.4.3 Shell window. The menu bar includes File, Edit, Shelf, Debug, Options, Window, and Help. The title bar says "Python 3.4.3 Shell". The shell area shows the following code and output:

```
Python 3.4.3 (default, Nov 17 2016, 01:08:31)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> x = "5"
>>> x
'5'
>>> x == 5
False
>>>
```

The line `x == 5` is highlighted with a red oval, and a red line points from this oval to the explanatory text on the right.

Inside a computer, a character is represented as a number (its "ASCII value")

the ASCII value of a digit is not the same as the digit itself:

$$'5' \neq 5$$

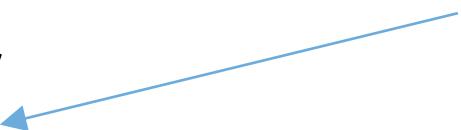
EXERCISE

```
>>> x = 27
```

```
>>> y = 'x'
```

```
>>> x == y
```

*What do you think will be
printed here?
Why?*



Basics of strings

```
>>> x = input()  
abcDE_fgHIJ_01234  
>>> x  
'abcDE_fgHIJ_01234'  
>>>  
>>> len(x)  
17  
>>> y = x.lower()  
>>> y  
'abcde_fghij_01234'  
>>>  
>>> x = y.upper()  
>> x  
'ABCDE_FGHIJ_01234'  
>>>
```

len(x) : length of a string x

Basics of strings

```
>>> x = input()  
abcDE_fgHIJ_01234  
>>> x  
'abcDE_fgHIJ_01234'  
>>>  
>>> len(x)  
17  
>>> y = x.lower()  
>>> y  
'abcde_fghij_01234'  
>>>  
>>> x = y.upper()  
>> x  
'ABCDE_FGHIJ_01234'  
>>>
```

len(x) : length of a string x

x.lower(), x.upper() : case conversion on the letters in a string x

- note that non-letter characters are not affected

Basics of strings

```
>>> x = input()  
abcDE_fgHIJ_01234  
>>> x  
'abcDE_fgHIJ_01234'  
>>>  
>>> len(x)  
17  
>>> y = x.lower()  
>>> y  
'abcde_fghij_01234'  
>>>  
>>> x = y.upper()  
>> x  
'ABCDE_FGHIJ_01234'  
>>>
```

len(x) : length of a string x

x.lower(), x.upper() : case conversion on the letters in a string x

- note that non-letter characters are not affected

Python supports a wide variety of string operations

- see www.tutorialspoint.com/python3/python_strings.htm

Basics of strings

```
>>> x = input()
```

```
abcdefghijklm
```

```
>>> x
```

```
'abcdefghijklm'
```

```
>>> x[3]
```

```
'd'
```

```
>>>
```

```
>>> x[3] = 'z'
```

Traceback (most recent call last):

File "<pyshell#193>", line 1, in <module>

 x[3] = 'z'

TypeError: 'str' object does not support item assignment

```
>>>
```

Basics of strings

```
>>> x = input()
```

```
abcdefghijklm
```

```
>>> x
```

```
'abcdefghijklm'
```

```
>>> x[3]
```

```
'd'
```

```
>>>
```

```
>>> x[3] = 'z'
```

strings are *immutable*, i.e., cannot be modified or updated

Traceback (most recent call last):

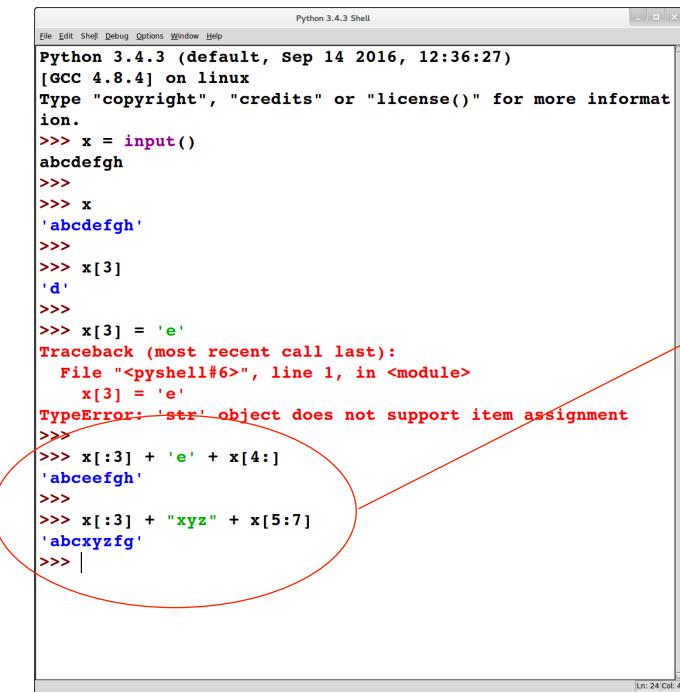
File "<pyshell#193>", line 1, in <module>

 x[3] = 'z'

TypeError: 'str' object does not support item assignment

```
>>>
```

Basics of strings



A screenshot of the Python 3.4.3 Shell window. The title bar says "Python 3.4.3 Shell". The menu bar includes File, Edit, Shelf, Debug, Options, Window, Help. The main area shows the following Python session:

```
Python 3.4.3 (default, Sep 14 2016, 12:36:27)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

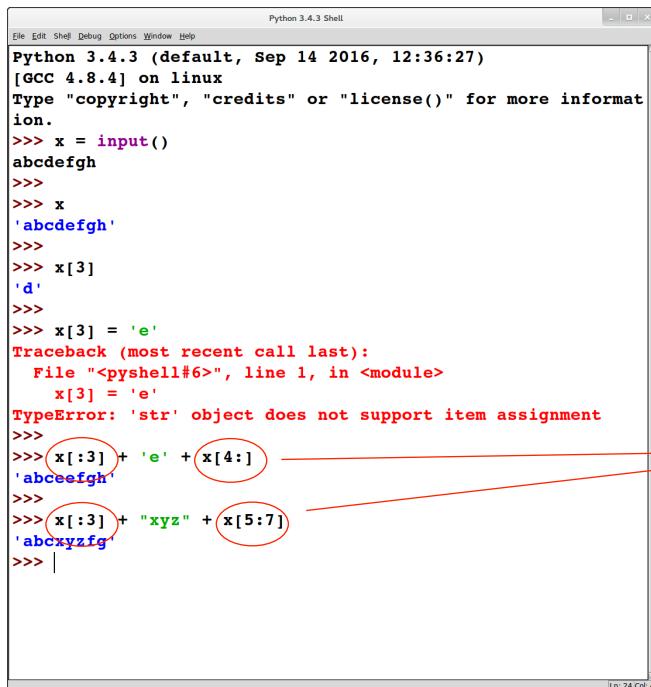
>>> x = input()
abcdefghijklm
>>>
>>> x
'abcdefghijklm'
>>>
>>> x[3]
'd'
>>>
>>> x[3] = 'e'
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    x[3] = 'e'
TypeError: 'str' object does not support item assignment
>>>
>>> x[:3] + 'e' + x[4:]
'abceefghijklm'
>>>
>>> x[:3] + "xyz" + x[5:7]
'abcxyzfgijklm'
>>> |
```

The line `x[3] = 'e'` is circled in red.

strings are immutable, i.e., cannot be modified or updated

to "modify" a string, we have to create a copy of it with the appropriate part(s) replaced by the new values

Basics of strings



The screenshot shows a Python 3.4.3 Shell window. The code demonstrates string slicing and concatenation. It starts by defining a string 'abcdefg' and attempting to modify its third character. This results in a `TypeError: 'str' object does not support item assignment`. Subsequent lines show how to create new strings by concatenating slices of the original string.

```
Python 3.4.3 (default, Sep 14 2016, 12:36:27)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> x = input()
abcdefg
>>>
>>> x
'abcdefg'
>>>
>>> x[3]
'd'
>>>
>>> x[3] = 'e'
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    x[3] = 'e'
TypeError: 'str' object does not support item assignment
>>>
>>> x[:3] + 'e' + x[4:]
'abceefgh'
>>>
>>> x[:3] + "xyz" + x[5:7]
'abcxyzfg'
>>> |
```

strings are immutable, i.e., cannot be modified or updated

to "modify" a string, we have to create a copy of it with the appropriate part(s) replaced by the new values

these operations are called "slicing"

Basics of strings

The screenshot shows a Python 3.4.3 Shell window. The code demonstrates various string operations:

```
Python 3.4.3 (default, Sep 14 2016, 12:36:27)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> x = input()
abcdefg
>>>
>>> x
'abcdefg'
>>>
>>> x[3]
'd'
>>>
>>> x[3] = 'e'
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    x[3] = 'e'
TypeError: 'str' object does not support item assignment
>>>
>>> x[:3] + 'e' + x[4:]
'abceefgh'
>>>
>>> x[:3] + "xyz" + x[5:7]
'abcxyzfg'
>>> |
```

Annotations with red ovals highlight the string slices `x[:3]`, `x[4:]`, and `x[5:7]`, which are concatenated with other strings.

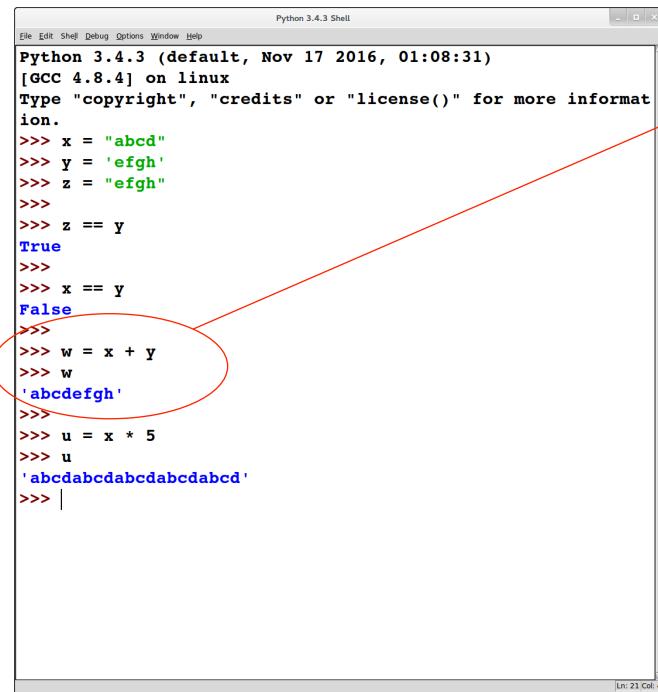
strings are immutable, i.e., cannot be modified or updated

to "modify" a string, we have to create a copy of it with the appropriate part(s) replaced by the new values

these operations are called "slicing"

+ applied to strings does concatenation

Basics of strings



A screenshot of the Python 3.4.3 Shell window. The title bar says "Python 3.4.3 Shell". The menu bar includes File, Edit, Shelf, Debug, Options, Window, and Help. The main area shows Python code and its output:

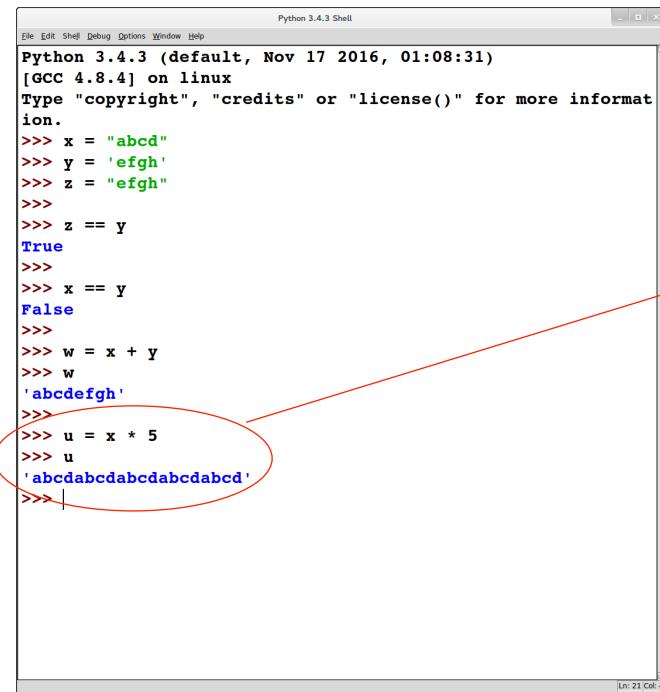
```
Python 3.4.3 (default, Nov 17 2016, 01:08:31)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> x = "abcd"
>>> y = 'efgh'
>>> z = "efgh"
>>>
>>> z == y
True
>>>
>>> x == y
False
>>> w = x + y
>>> w
'abcdefgh'
>>>
>>> u = x * 5
>>> u
'abcdabcdabcdabcdabcd'
>>> |
```

The line `w = x + y` is highlighted with a red oval. The output `'abcdefgh'` is also highlighted with a red oval.

+ applied to strings does concatenation

Basics of strings



A screenshot of the Python 3.4.3 Shell window. The menu bar includes File, Edit, Shelf, Debug, Options, Window, and Help. The title bar says "Python 3.4.3 Shell". The code area shows:

```
Python 3.4.3 (default, Nov 17 2016, 01:08:31)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> x = "abcd"
>>> y = 'efgh'
>>> z = "efgh"
>>>
>>> z == y
True
>>>
>>> x == y
False
>>> w = x + y
>>> w
'abcdefghijklm'
>>>
>>> u = x * 5
>>> u
'abcdaabcdabcdabcdabcd'
```

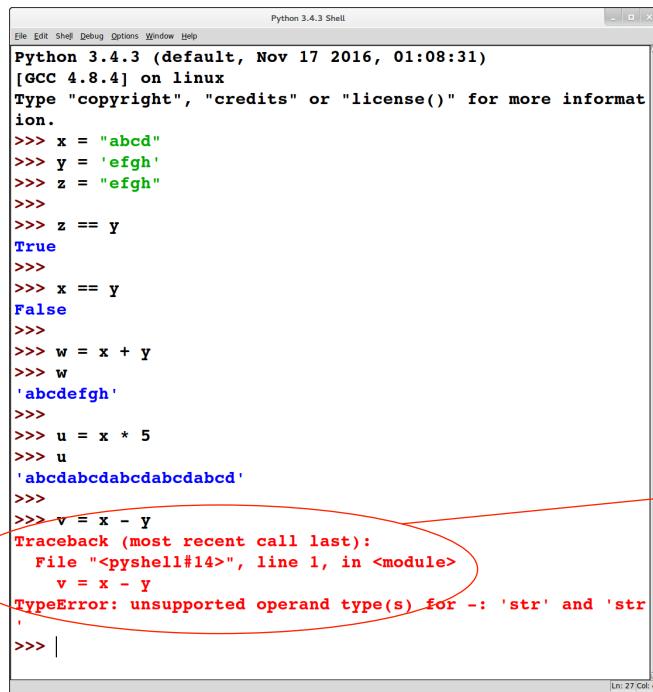
The line `u = x * 5` is highlighted with a red oval.

+ applied to strings does concatenation

'*' applied to strings:

- does repeated concatenation *if one argument is a number*
- generates an error otherwise

Basics of strings



The screenshot shows a Python 3.4.3 Shell window. The code entered is:

```
Python 3.4.3 (default, Nov 17 2016, 01:08:31)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> x = "abcd"
>>> y = 'efgh'
>>> z = "efgh"
>>>
>>> z == y
True
>>>
>>> x == y
False
>>>
>>> w = x + y
>>> w
'abcdefghijklm'
>>>
>>> u = x * 5
>>> u
'aaaaaaaaaa'
>>>
>>> v = x - y
Traceback (most recent call last):
  File "<pyshell#14>", line 1, in <module>
    v = x - y
TypeError: unsupported operand type(s) for -: 'str' and 'str'
>>> |
```

A red oval highlights the line `v = x - y` and the resulting traceback.

+ applied to strings does concatenation

* applied to strings:

- does repeated concatenation *if one argument is a number*
- generates an error otherwise

not all arithmetic operators carry over to strings

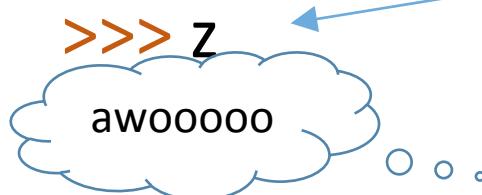
EXERCISE

```
>>> x = "whoa!"
```

```
>>> y = x[2] * len(x)
```

```
>>> z = x[3] + x[0] + y
```

what is printed here?



EXERCISE

```
>>> x = input()  
>>> y = x + x  
>>> int(x) == int(y)
```

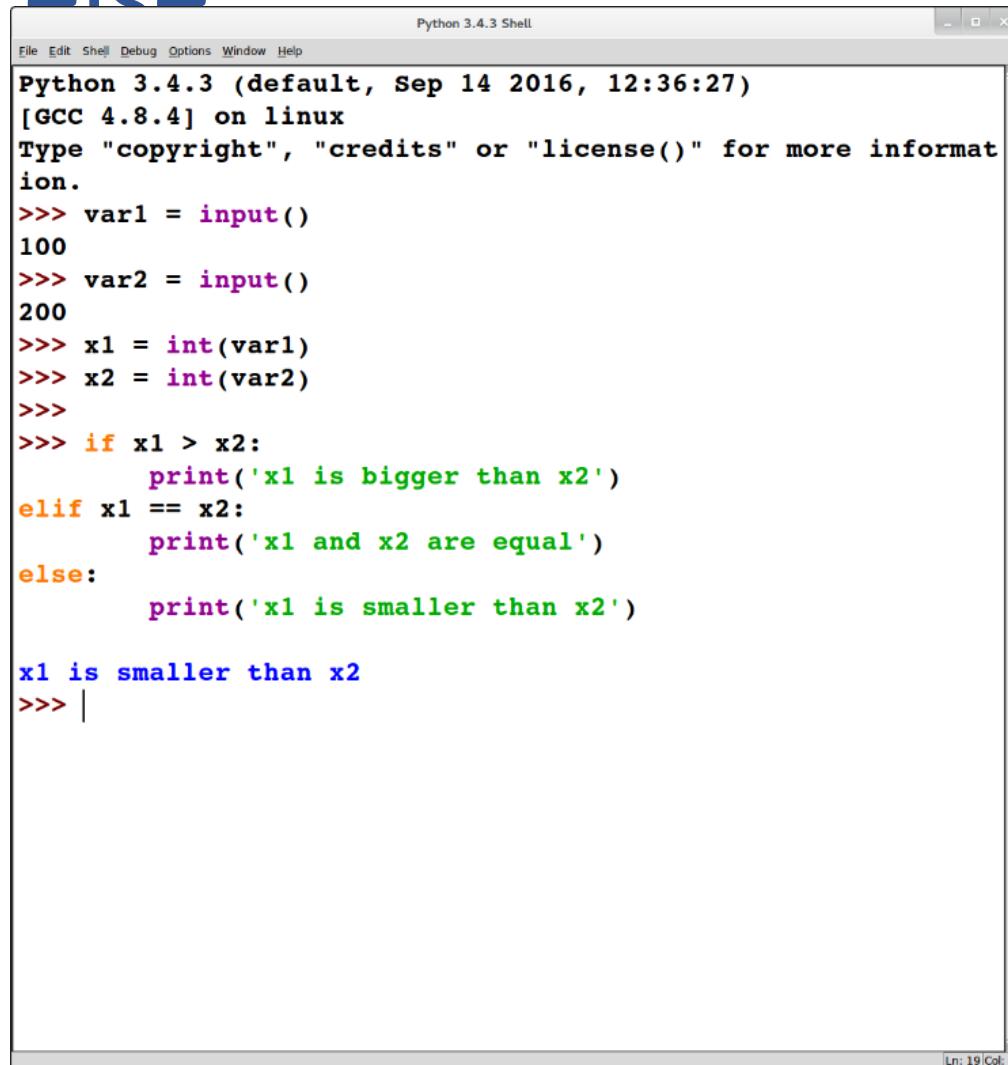
True

*what input value(s) will cause
this to work as shown?*



python review: conditionals

Conditional statements: if/elif/ else



The screenshot shows a window titled "Python 3.4.3 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main area displays the following Python code and its execution:

```
Python 3.4.3 (default, Sep 14 2016, 12:36:27)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> var1 = input()
100
>>> var2 = input()
200
>>> x1 = int(var1)
>>> x2 = int(var2)
>>>
>>> if x1 > x2:
...     print('x1 is bigger than x2')
elif x1 == x2:
    print('x1 and x2 are equal')
else:
    print('x1 is smaller than x2')

x1 is smaller than x2
>>> |
```

The status bar at the bottom right indicates "Ln: 19 Col: 4".

Conditional statements: if/elif/else

```
>>> var1 = input()  
100  
>>> var2 = input()  
200  
>>> x1 = int(var1)  
>>> x2 = int(var2)  
>>>  
>>> if x1 > x2:  
        print('x1 is bigger than x2')  
elif x1 == x2:  
    print('x1 and x2 are equal')  
else:  
    print('x1 is smaller than x2')  
  
x1 is smaller than x2  
>>>
```

- **if-statement syntax:**

```
if BooleanExpr :  
    stmt  
    ...  
elif BooleanExpr :  
    stmt  
    ...  
elif ...  
    ...  
else:  
    stmt  
    ...
```

elifs are optional
(use as needed)

Conditional statements: if/elif/else

```
>>> var1 = input()
100
>>> var2 = input()
200
>>> x1 = int(var1)
>>> x2 = int(var2)
>>>
>>> if x1 > x2:
        print('x1 is bigger than x2')
    elif x1 == x2:
        print('x1 and x2 are equal')
    else:
        print('x1 is smaller than
x2')
x1 is smaller than x2
>>>
```

- **if**-statement syntax:

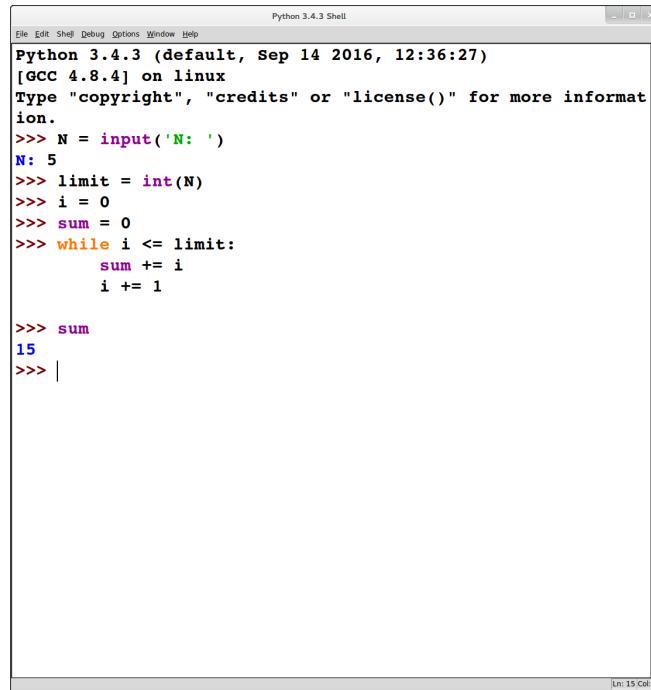
```
if BooleanExpr :
    stmt
    ...
elif BooleanExpr :
    stmt
    ...
elif ... :
    ...
else:
    stmt
    ...
```

} **elifs** are optional
(use as needed)

} **else** is optional

python review: while loops

Loops I: while



The screenshot shows a window titled "Python 3.4.3 Shell". The menu bar includes "File", "Edit", "Shell", "Debug", "Options", "Window", and "Help". The main area displays the following Python code and its execution:

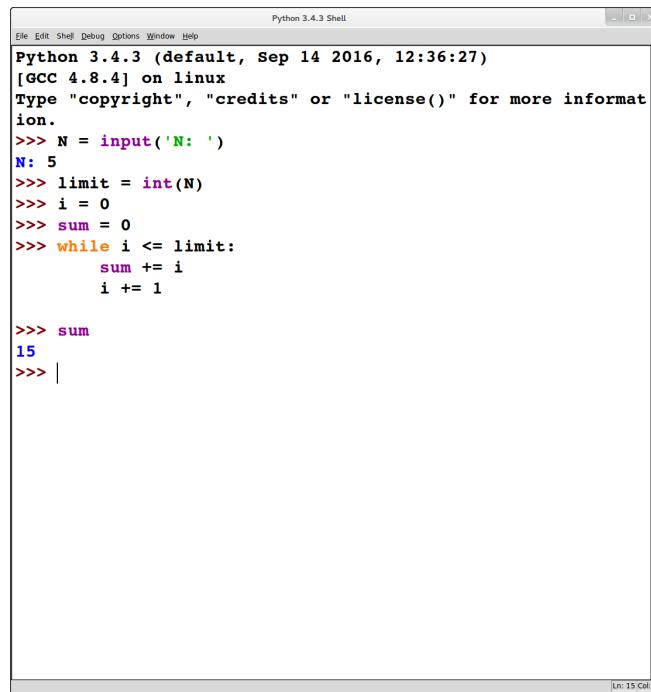
```
Python 3.4.3 (default, Sep 14 2016, 12:36:27)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> N = input('N: ')
N: 5
>>> limit = int(N)
>>> i = 0
>>> sum = 0
>>> while i <= limit:
...     sum += i
...     i += 1

>>> sum
15
>>> |
```

The status bar at the bottom right indicates "Ln: 15 Col: 4".

Loops I: while



A screenshot of the Python 3.4.3 Shell window. The title bar says "Python 3.4.3 Shell". The menu bar includes File, Edit, Shelf, Debug, Options, Window, and Help. The main window displays the following Python code:

```
Python 3.4.3 (default, Sep 14 2016, 12:36:27)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> N = input('N: ')
N: 5
>>> limit = int(N)
>>> i = 0
>>> sum = 0
>>> while i <= limit:
...     sum += i
...     i += 1

>>> sum
15
>>> |
```

The status bar at the bottom right shows "Ln: 15 Col: 4".

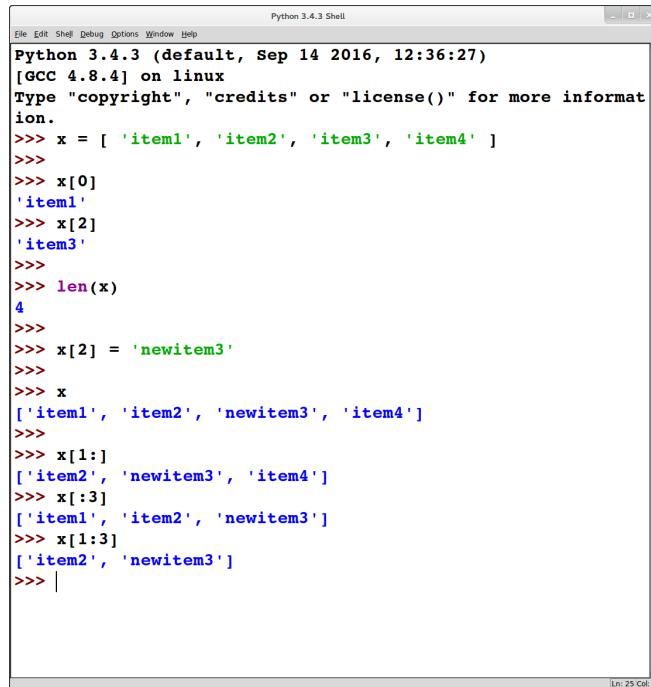
- **while**-statement syntax:

```
while BooleanExpr :  
    stmt1  
    ...  
    stmtn
```

- $stmt_1 \dots stmt_n$ are executed repeatedly as long as *BooleanExpr* is True

python review: lists (aka arrays)

Lists



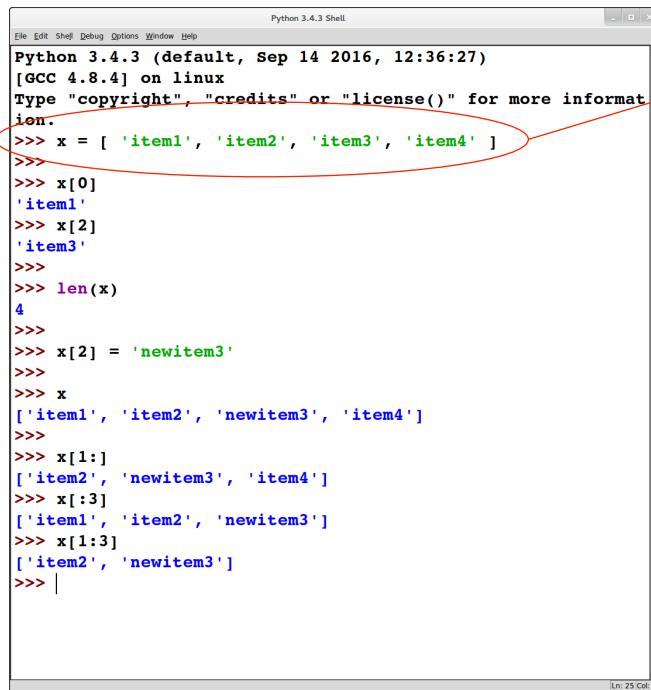
The screenshot shows a window titled "Python 3.4.3 Shell". The menu bar includes "File", "Edit", "Shell", "Debug", "Options", "Window", and "Help". The main area displays a Python session:

```
Python 3.4.3 (default, Sep 14 2016, 12:36:27)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> x = [ 'item1', 'item2', 'item3', 'item4' ]
>>>
>>> x[0]
'item1'
>>> x[2]
'item3'
>>>
>>> len(x)
4
>>>
>>> x[2] = 'newitem3'
>>>
>>> x
['item1', 'item2', 'newitem3', 'item4']
>>>
>>> x[1:]
['item2', 'newitem3', 'item4']
>>> x[:3]
['item1', 'item2', 'newitem3']
>>> x[1:3]
['item2', 'newitem3']
>>> |
```

At the bottom right of the shell window, there is a status bar with the text "[Ln: 25 Col: 4]".

Lists



A screenshot of the Python 3.4.3 Shell window. The title bar says "Python 3.4.3 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, Help. The main area shows Python code and its output:

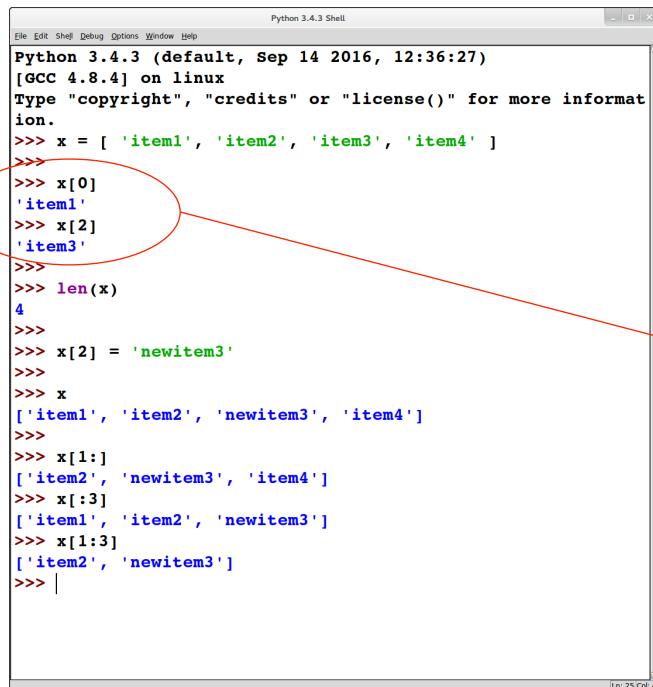
```
Python 3.4.3 (default, Sep 14 2016, 12:36:27)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> x = [ 'item1', 'item2', 'item3', 'item4' ]
>>>
>>> x[0]
'item1'
>>> x[2]
'item3'
>>>
>>> len(x)
4
>>>
>>> x[2] = 'newitem3'
>>>
>>> x
['item1', 'item2', 'newitem3', 'item4']
>>>
>>> x[1:]
['item2', 'newitem3', 'item4']
>>> x[:3]
['item1', 'item2', 'newitem3']
>>> x[1:3]
['item2', 'newitem3']
>>> |
```

The line "x = ['item1', 'item2', 'item3', 'item4']" has a red oval drawn around it, and a red arrow points from this oval to the text "a list (or array) is a sequence of values" located to the right of the shell window.

a list (or array) is a sequence of values

Lists



```
Python 3.4.3 (default, Sep 14 2016, 12:36:27)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> x = [ 'item1', 'item2', 'item3', 'item4' ]
>>>
>>> x[0]
'item1'
>>> x[2]
'item3'
>>>
>>> len(x)
4
>>>
>>> x[2] = 'newitem3'
>>>
>>> x
['item1', 'item2', 'newitem3', 'item4']
>>>
>>> x[1:]
['item2', 'newitem3', 'item4']
>>> x[:3]
['item1', 'item2', 'newitem3']
>>> x[1:3]
['item2', 'newitem3']
>>> |
```

a list (or array) is a sequence of values

accessing list elements (i.e., indexing),
computing length: similar to strings

- non-negative index values (≥ 0) index from the front of the list
 - the first element has index 0
- negative index values index from the end of the list
 - the last element has index -1

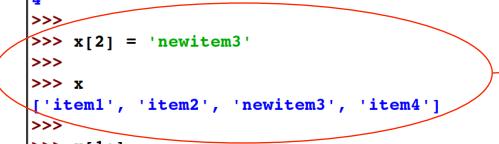
EXERCISE

```
>>> x = [ "abc", "def", "ghi", "jkl" ]
```

```
>>> x[1] + x[-1]
```

what do you think will be printed here?

Lists



```
Python 3.4.3 (default, Sep 14 2016, 12:36:27)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> x = [ 'item1', 'item2', 'item3', 'item4' ]
>>>
>>> x[0]
'item1'
>>> x[2]
'item3'
>>>
>>> len(x)
4
>>>
>>> x[2] = 'newitem3'
>>>
>>> x
['item1', 'item2', 'newitem3', 'item4']
>>>
>>> x[1:]
['item2', 'newitem3', 'item4']
>>> x[:3]
['item1', 'item2', 'newitem3']
>>> x[1:3]
['item2', 'newitem3']
>>> |
```

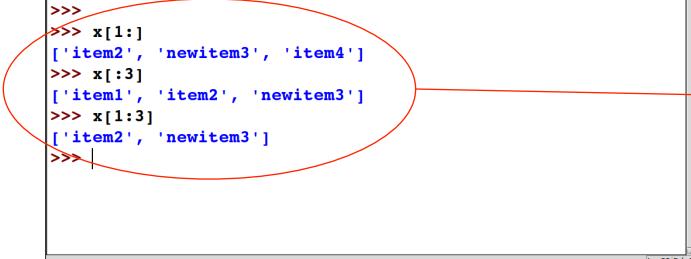
a list (or array) is a sequence of values

accessing list elements (i.e., indexing),
computing length: similar to strings

lists are mutable, i.e., can be modified
or updated

- different from strings

Lists



```
Python 3.4.3 (default, Sep 14 2016, 12:36:27)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> x = [ 'item1', 'item2', 'item3', 'item4' ]
>>>
>>> x[0]
'item1'
>>> x[2]
'item3'
>>>
>>> len(x)
4
>>>
>>> x[2] = 'newitem3'
>>>
>>> x
['item1', 'item2', 'newitem3', 'item4']
>>>
>>> x[1:]
['item2', 'newitem3', 'item4']
>>> x[:3]
['item1', 'item2', 'newitem3']
>>> x[1:3]
['item2', 'newitem3']
>>>
```

a list (or array) is a sequence of values

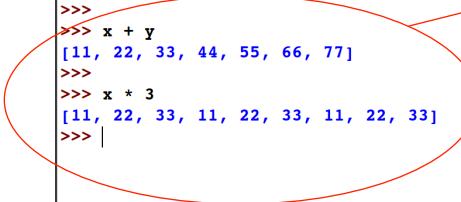
accessing list elements (i.e., indexing),
computing length: similar to strings

lists are mutable, i.e., can be modified
or updated

- different from strings

slicing : similar to strings

Lists

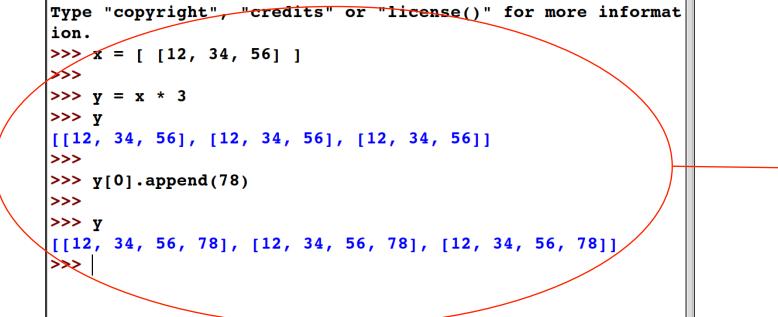


```
Python 3.4.3 (default, Nov 17 2016, 01:08:31)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> x = [11, 22, 33]
>>> y = [44, 55, 66, 77]
>>>
>>> x + y
[11, 22, 33, 44, 55, 66, 77]
>>>
>>> x * 3
[11, 22, 33, 11, 22, 33, 11, 22, 33]
>>> |
```

concatenation (+ and *) : similar to strings

Lists



```
Python 3.4.3 (default, Nov 17 2016, 01:08:31)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> x = [ 12, 34, 56 ]
>>>
>>> y = x * 3
>>> y
[[12, 34, 56], [12, 34, 56], [12, 34, 56]]
>>>
>>> y[0].append(78)
>>>
>>> y
[[12, 34, 56, 78], [12, 34, 56, 78], [12, 34, 56, 78]]
>>> |
```

concatenation (+ and *) : similar to strings

- these operators create “shallow” copies
- due to list mutability, this can cause unexpected behavior

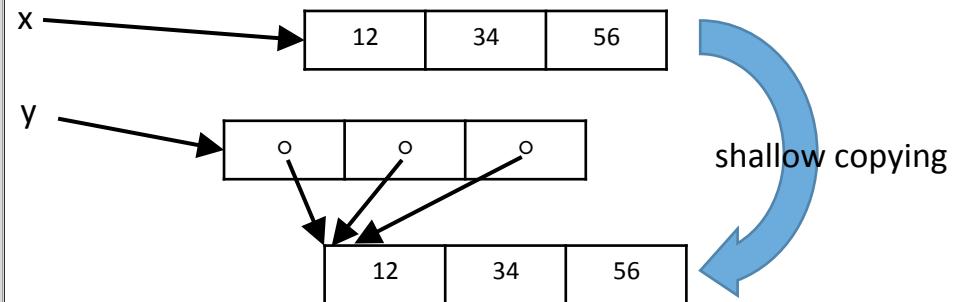
Lists

```
Python 3.4.3 (default, Nov 17 2016, 01:08:31)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> x = [ 12, 34, 56 ]
>>>
>>> y = x * 3
>>> y
[[12, 34, 56], [12, 34, 56], [12, 34, 56]]
>>>
>>> y[0].append(78)
>>>
>>> y
[[12, 34, 56, 78], [12, 34, 56, 78], [12, 34, 56, 78]]
>>> |
```

concatenation (+ and *) : similar to strings

- these operators create “shallow” copies
- due to list mutability, this can cause unexpected behavior



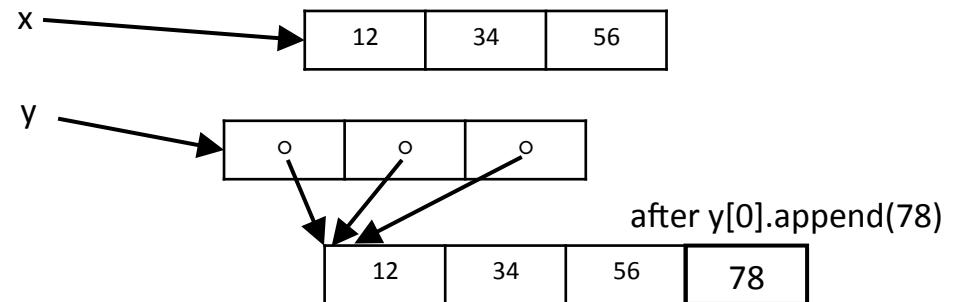
Lists

```
Python 3.4.3 (default, Nov 17 2016, 01:08:31)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

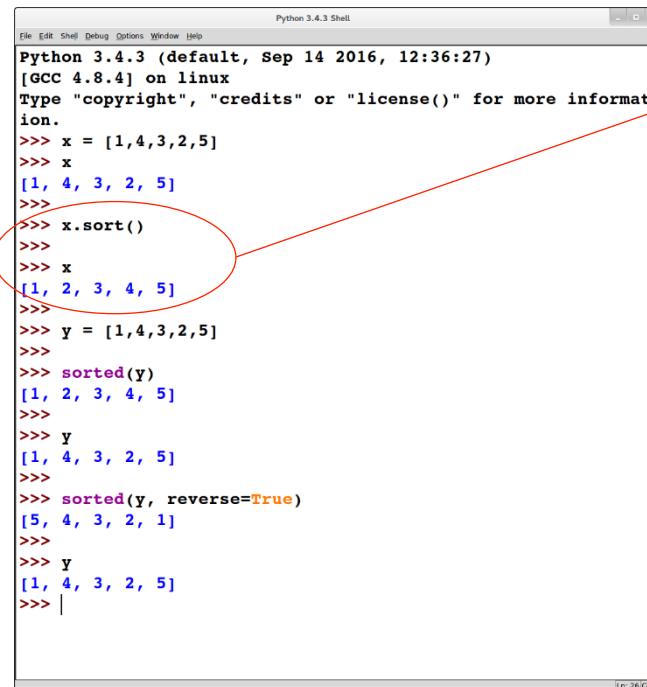
>>> x = [ 12, 34, 56 ]
>>>
>>> y = x * 3
>>> y
[[12, 34, 56], [12, 34, 56], [12, 34, 56]]
>>>
>>> y[0].append(78)
>>>
>>> y
[[12, 34, 56, 78], [12, 34, 56, 78], [12, 34, 56, 78]]
>>> |
```

concatenation (+ and *) : similar to strings

- these operators create “shallow” copies
- due to list mutability, this can cause unexpected behavior



Lists: sorting

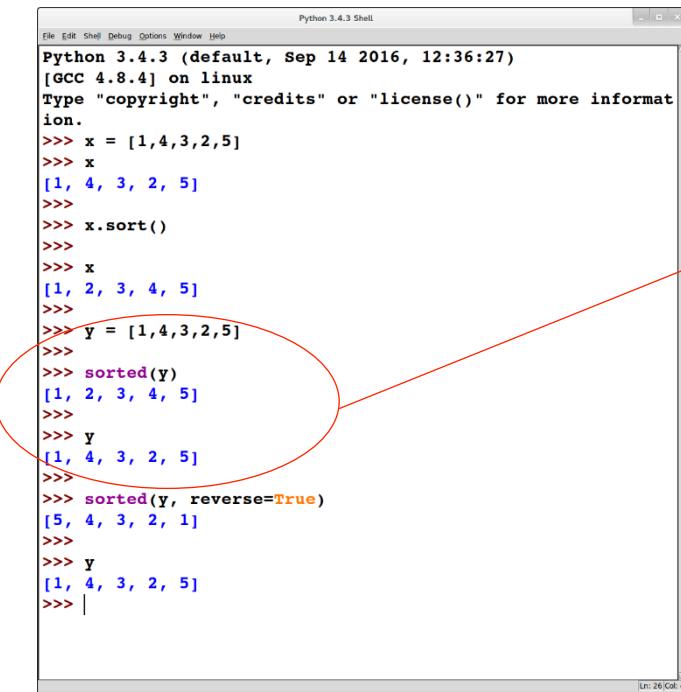


```
Python 3.4.3 (default, Sep 14 2016, 12:36:27)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> x = [1,4,3,2,5]
>>> x
[1, 4, 3, 2, 5]
>>> x.sort()
>>>
>>> x
[1, 2, 3, 4, 5]
>>>
>>> y = [1,4,3,2,5]
>>>
>>> sorted(y)
[1, 2, 3, 4, 5]
>>>
>>> y
[1, 4, 3, 2, 5]
>>>
>>> sorted(y, reverse=True)
[5, 4, 3, 2, 1]
>>>
>>> y
[1, 4, 3, 2, 5]
>>> |
```

sort() : sorts a list

Lists: sorting



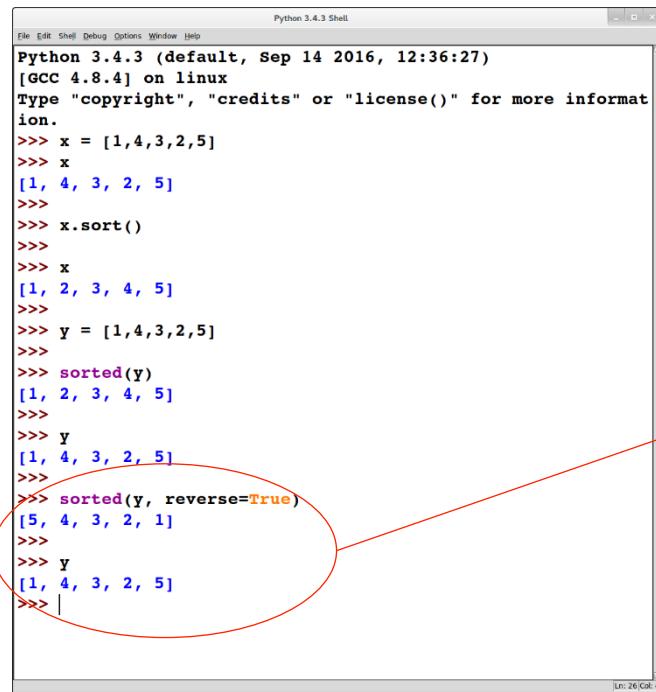
```
Python 3.4.3 (default, Sep 14 2016, 12:36:27)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> x = [1,4,3,2,5]
>>> x
[1, 4, 3, 2, 5]
>>>
>>> x.sort()
>>>
>>> x
[1, 2, 3, 4, 5]
>>>
>>> y = [1,4,3,2,5]
>>>
>>> sorted(y)
[1, 2, 3, 4, 5]
>>>
>>> y
[1, 4, 3, 2, 5]
>>>
>>> sorted(y, reverse=True)
[5, 4, 3, 2, 1]
>>>
>>> y
[1, 4, 3, 2, 5]
>>> |
```

sort() : sorts a list

sorted() : creates a sorted copy of a list;
the original list is not changed

Lists: sorting



A screenshot of the Python 3.4.3 Shell window. The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The title bar says "Python 3.4.3 Shell". The main area shows the following Python session:

```
Python 3.4.3 (default, Sep 14 2016, 12:36:27)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> x = [1,4,3,2,5]
>>> x
[1, 4, 3, 2, 5]
>>>
>>> x.sort()
>>>
>>> x
[1, 2, 3, 4, 5]
>>>
>>> y = [1,4,3,2,5]
>>>
>>> sorted(y)
[1, 2, 3, 4, 5]
>>>
>>> y
[1, 4, 3, 2, 5]
>>>
>>> sorted(y, reverse=True)
[5, 4, 3, 2, 1]
>>>
>>> y
[1, 4, 3, 2, 5]
>>> |
```

The command `x.sort()` is highlighted with a red oval.

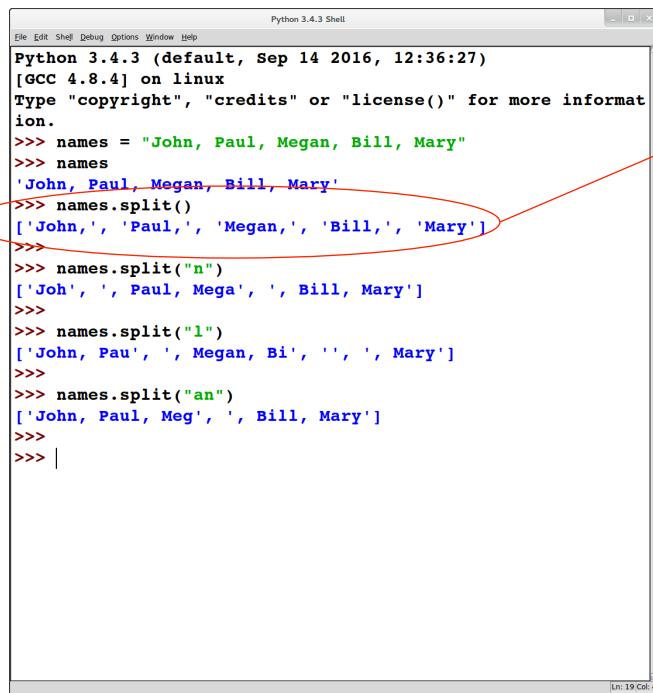
`sort()` : sorts a list

`sorted()` : creates a sorted copy of a list;
the original list is not changed

the optional argument `reverse` specifies
ascending or descending order

python review: lists \leftrightarrow strings

Strings → lists



A screenshot of the Python 3.4.3 Shell window. The code shown is:

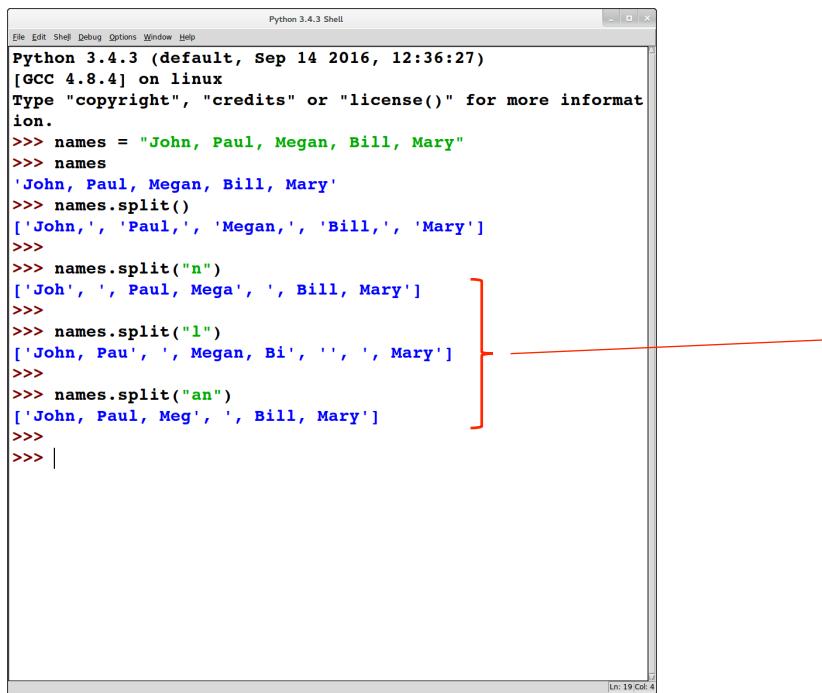
```
Python 3.4.3 (default, Sep 14 2016, 12:36:27)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> names = "John, Paul, Megan, Bill, Mary"
>>> names
'John, Paul, Megan, Bill, Mary'
>>> names.split()
['John', 'Paul', 'Megan', 'Bill', 'Mary']
>>>
>>> names.split("\n")
['Joh', ' ', 'Paul', ' ', 'Mega', ' ', 'Bill', 'Mary']
>>>
>>> names.split("l")
['John', 'Pau', ' ', 'Megan', 'Bi', ' ', ' ', 'Mary']
>>>
>>> names.split("an")
['John', 'Paul', 'Meg', ' ', 'Bill', 'Mary']
>>>
>>> |
```

The line `names.split()` and its resulting list are circled in red.

split() : splits a string on whitespace
returns a list of strings

Strings → lists



A screenshot of the Python 3.4.3 Shell window. The title bar says "Python 3.4.3 Shell". The window content shows the following Python session:

```
Python 3.4.3 (default, Sep 14 2016, 12:36:27)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> names = "John, Paul, Megan, Bill, Mary"
>>> names
'John, Paul, Megan, Bill, Mary'
>>> names.split()
['John', 'Paul', 'Megan', 'Bill', 'Mary']
>>>
>>> names.split("\n")
['Joh', ' ', 'Paul', ' ', 'Mega', ' ', 'Bill', ' ', 'Mary']
>>>
>>> names.split("l")
['John', 'Pau', ' ', 'Megan', 'Bi', ' ', ' ', 'Mary']
>>>
>>> names.split("an")
['John', 'Paul', 'Meg', ' ', 'Bill', 'Mary']
>>>
>>> |
```

The code demonstrates the `split()` method on a string. It first splits the string by whitespace, then by a newline character, and finally by the letter 'l'. A red bracket on the right side of the screen groups the last two examples together, with a red arrow pointing from the text "split(*delim*) : given an optional argument" to this bracket.

split() : splits a string on whitespace
returns a list of strings

split(*delim*) : given an optional
argument

delim, splits the string on
delim

Lists → strings

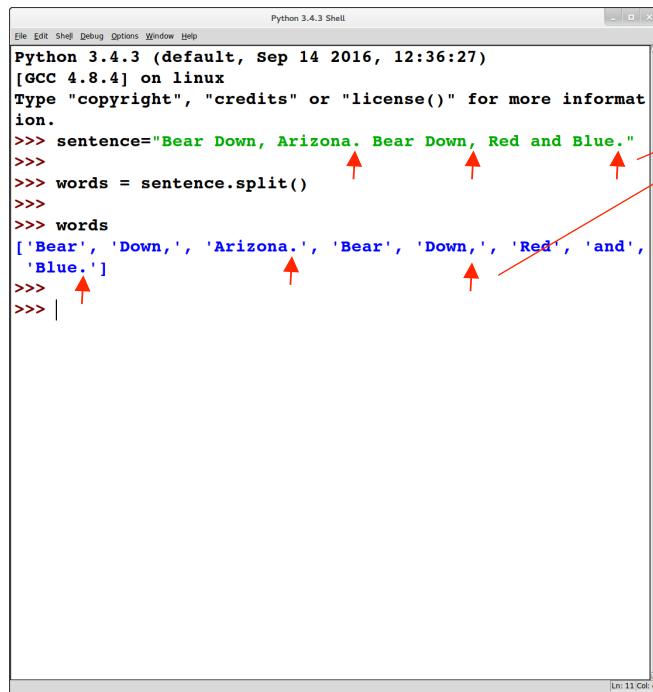
```
Python 3.4.3 (default, Sep 14 2016, 12:36:27)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> names = "John Paul Megan Bill Mary"
>>> x = names.split()
>>>
>>> x
['John', 'Paul', 'Megan', 'Bill', 'Mary']
>>>
>>> "-".join(x)
'John-Paul-Megan-Bill-Mary'
>>>
>>> "^.^".join(x)
'John^.^Paul^.^Megan^.^Bill^.^Mary'
>>>
>>> "XYZ".join(x)
'JohnXYZPaulXYZMeganXYZBillXYZMary'
>>>
>>> |
```

delim.join(list) : joins the strings in *list* using the string *delim* as the delimiter

returns a string

String trimming



A screenshot of the Python 3.4.3 Shell window. The title bar says "Python 3.4.3 Shell". The menu bar includes File, Edit, Shelf, Debug, Options, Window, and Help. The main area shows the following Python code:

```
Python 3.4.3 (default, Sep 14 2016, 12:36:27)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> sentence="Bear Down, Arizona. Bear Down, Red and Blue."
>>> words = sentence.split()
>>>
>>> words
['Bear', 'Down,', 'Arizona.', 'Bear', 'Down,', 'Red', 'and',
 'Blue.']
>>> |
```

The output is displayed in blue. Several red arrows point from the text "what if we wanted to get rid of the punctuation?" to the punctuation marks in the string (" ", ".", ",", and "-") and in the resulting list elements (",", ".").

what if we wanted to get rid of the punctuation?

String trimming



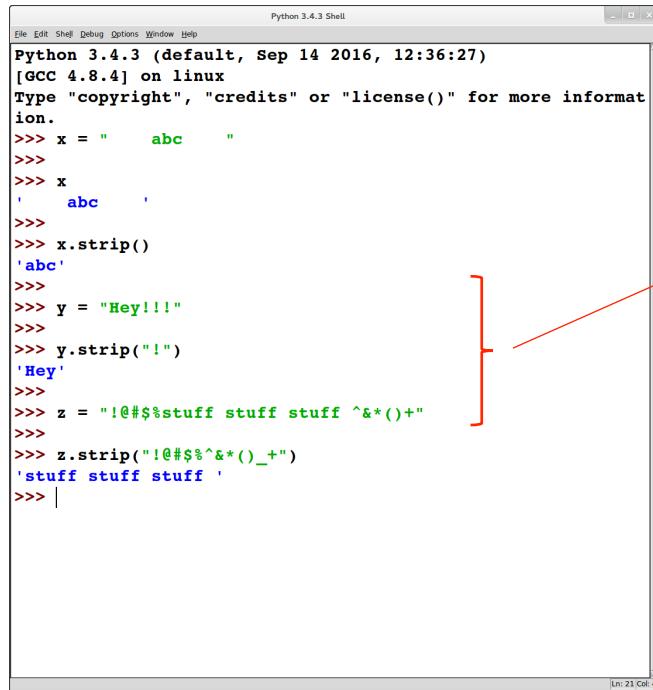
```
Python 3.4.3 (default, Sep 14 2016, 12:36:27)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> x = " abc "
>>> x
' abc '
>>>
>>> x.strip()
'abc'
>>>
>>> y = "Hey!!!"
>>>
>>> y.strip("!")
'Hey'
>>>
>>> z = "!@#$stuff stuff stuff ^&*()_+"
>>>
>>> z.strip("!@#$%^&*()_+")
'stuff stuff stuff '
```

x.strip() : removes whitespace from either end of the string x

returns a string

String trimming



A screenshot of the Python 3.4.3 Shell window. The code examples demonstrate the use of the `strip()` method:

```
Python 3.4.3 (default, Sep 14 2016, 12:36:27)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

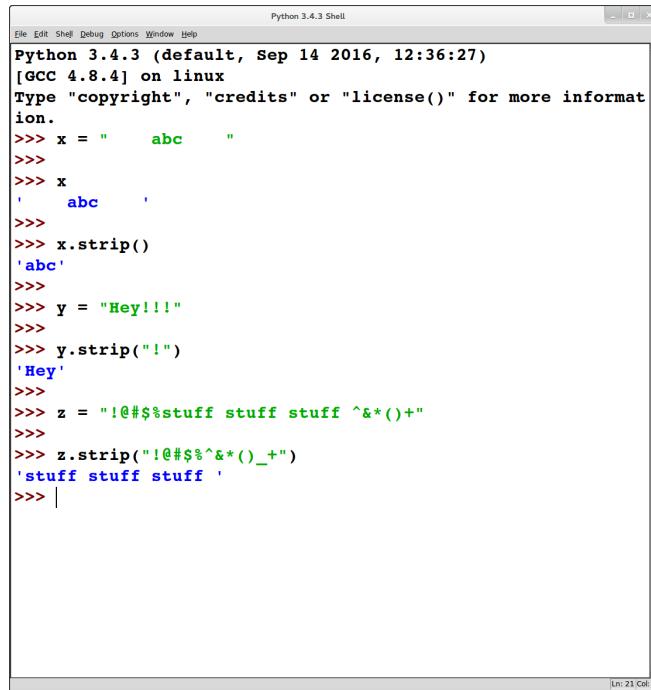
>>> x = "      abc      "
>>>
>>> x
'      abc      '
>>>
>>> x.strip()
'abc'
>>>
>>> y = "Hey!!!"
>>>
>>> y.strip("!")
'Hey'
>>>
>>> z = "!@#$stuff stuff stuff ^&*()+"
>>>
>>> z.strip("!@#$%^&*()_+")
'stuff stuff stuff '
>>> |
```

The code shows how `x.strip()` removes whitespace from both ends of the string `x`, and how `y.strip("!")` removes the exclamation mark from both ends of the string `y`. A red bracket on the right side of the slide groups the first two examples, pointing to the explanatory text below.

`x.strip()` : removes whitespace from either end of the string `x`

`x.strip(string)` : given an optional argument `string`, removes any character in `string` from either end of `x`

String trimming



A screenshot of the Python 3.4.3 Shell window. The title bar says "Python 3.4.3 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, Help. The main area shows the following Python code:

```
Python 3.4.3 (default, Sep 14 2016, 12:36:27)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> x = "      abc      "
>>>
>>> x
'      abc      '
>>>
>>> x.strip()
'abc'
>>>
>>> y = "Hey!!!"
>>>
>>> y.strip("!")
'Hey'
>>>
>>> z = "!@#$stuff stuff stuff ^&*()_+"
>>>
>>> z.strip("!@#$%^&*()_+")
'stuff stuff stuff '
```

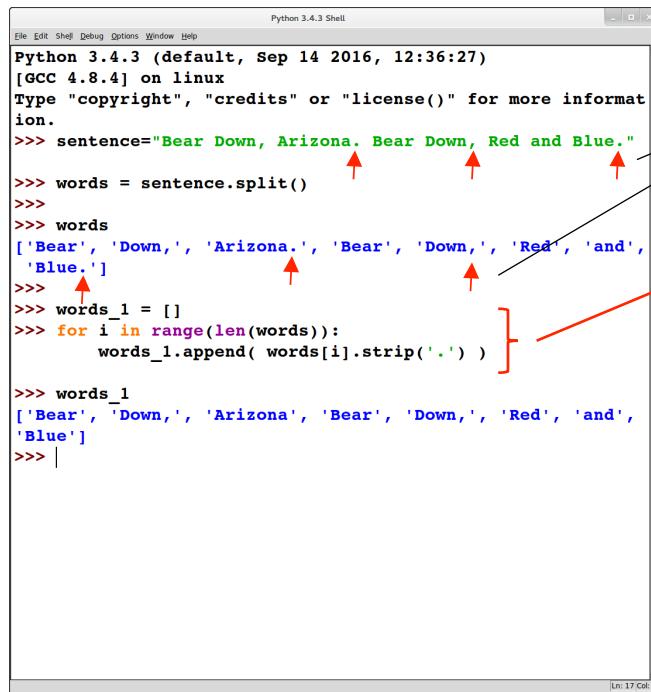
The status bar at the bottom right says "Ln: 21 Col: 4".

`x.strip()` : removes whitespace from either end of the string `x`

`x.strip(string)` : given an optional argument `string`, removes any character in `string` from either end of `x`

`rstrip()`, `lstrip()` : similar to `strip()` but trims from one end of the string

String trimming



```
Python 3.4.3 (default, Sep 14 2016, 12:36:27)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> sentence="Bear Down, Arizona. Bear Down, Red and Blue."
>>> words = sentence.split()
>>>
>>> words
['Bear', 'Down,', 'Arizona.', 'Bear', 'Down,', 'Red', 'and',
 'Blue.']
>>> words_1 = []
>>> for i in range(len(words)):
...     words_1.append( words[i].strip('.'))
...
>>> words_1
['Bear', 'Down,', 'Arizona', 'Bear', 'Down,', 'Red', 'and',
 'Blue']
>>> |
```

what if we wanted to get rid of the punctuation?

- iterate over the list
- use strip() to trim each word in the list
- reassemble the trimmed words into a list

String trimming

```
Python 3.4.3 (default, Sep 14 2016, 12:36:27)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

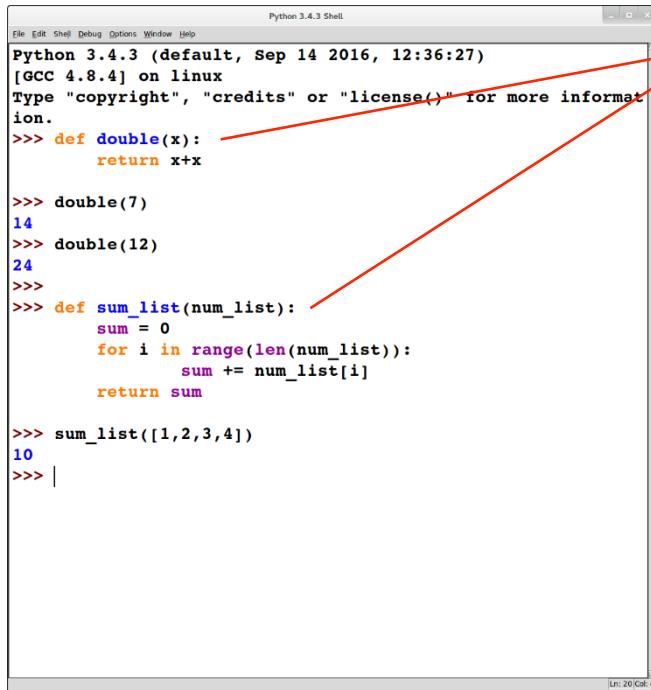
>>> sentence="Bear Down, Arizona. Bear Down, Red and Blue."
>>>
>>> words = sentence.split()
>>>
>>> words
['Bear', 'Down', 'Arizona.', 'Bear', 'Down', 'Red', 'and',
 'Blue.']
>>> words_1 = []
>>> for i in range(len(words)):
    words_1.append( words[i].strip('.'))
    ]
>>> words_1
['Bear', 'Down', 'Arizona', 'Bear', 'Down', 'Red', 'and',
 'Blue']
>>>
>>> words_2 = []
>>> i = 0
>>> while i < len(words):
    words_2 += [ words[i].strip(',') ]
    i += 1
    ]
>>> words_2
['Bear', 'Down', 'Arizona', 'Bear', 'Down', 'Red', 'and', 'B
lue']
>>> |
```

what if we wanted to get rid of the punctuation?

- iterate over the list
- use strip() to trim each word in the list
- reassemble the trimmed words into a list

python review: functions

Functions



A screenshot of the Python 3.4.3 Shell window. The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The title bar says "Python 3.4.3 Shell". The shell window displays the following Python code:

```
Python 3.4.3 (default, Sep 14 2016, 12:36:27)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> def double(x):
...     return x+x

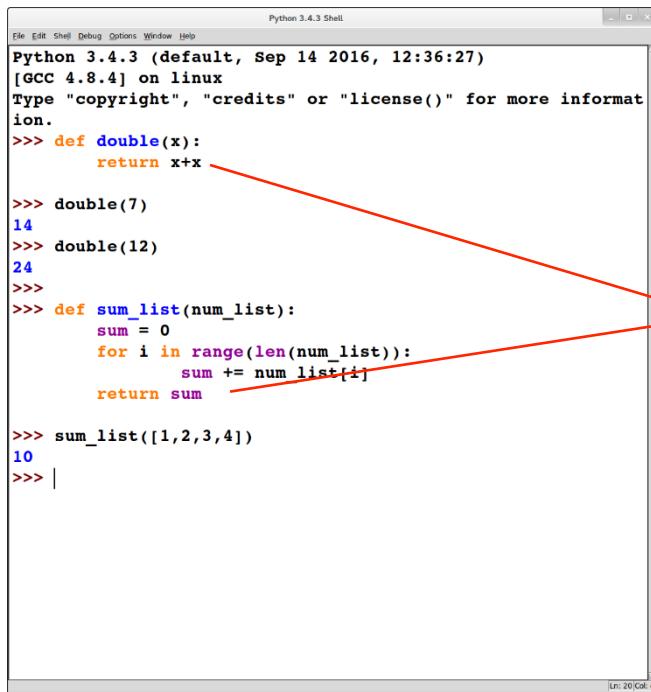
>>> double(7)
14
>>> double(12)
24
>>>
>>> def sum_list(num_list):
...     sum = 0
...     for i in range(len(num_list)):
...         sum += num_list[i]
...     return sum

>>> sum_list([1,2,3,4])
10
>>> |
```

The code defines two functions: `double` and `sum_list`. The `double` function takes one argument `x` and returns its double. The `sum_list` function takes a list of numbers and returns their sum. Both functions are tested with the values 7 and 12 respectively. A red arrow points from the first bullet point below to the `def double(x):` line.

- **`def fn_name (arg1 , ..., argn)`**
 - defines a function *fn_name* with *n* arguments *arg₁* , ..., *arg_n*

Functions



The screenshot shows a Python 3.4.3 Shell window. The code defines two functions: `double` which returns the square of its argument, and `sum_list` which calculates the sum of all elements in a list. The `double` function is demonstrated with arguments 7 and 12, both resulting in 14. The `sum_list` function is demonstrated with the list [1, 2, 3, 4], resulting in 10. Red arrows point from the text "with n arguments arg_1, \dots, arg_n " to the parameter list in the `def double` line, and from the text "returns the value of the expression $expr$ to the caller" to the `return sum` line in the `sum_list` definition.

```
Python 3.4.3 (default, Sep 14 2016, 12:36:27)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

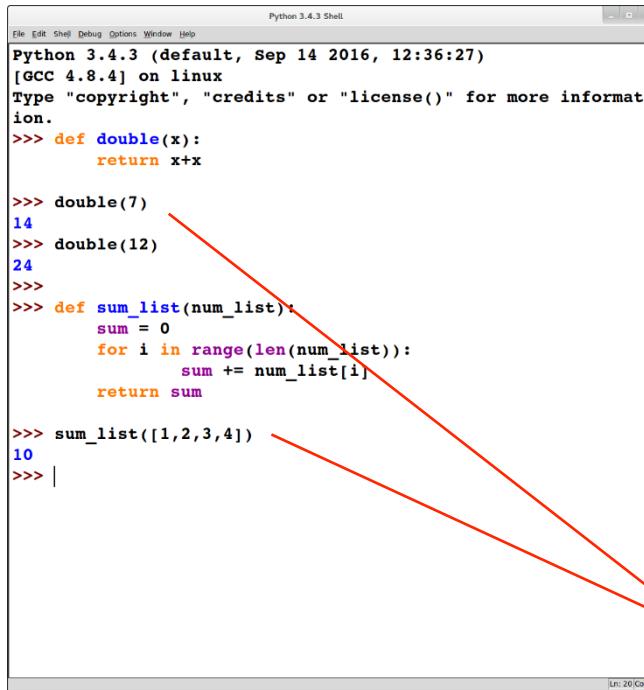
>>> def double(x):
...     return x*x

>>> double(7)
14
>>> double(12)
24
>>>
>>> def sum_list(num_list):
...     sum = 0
...     for i in range(len(num_list)):
...         sum += num_list[i]
...     return sum

>>> sum_list([1,2,3,4])
10
>>> |
```

- **`def fn_name (arg1 , ..., argn)`**
 - defines a function fn_name with n arguments arg_1, \dots, arg_n
- **`return expr`**
 - optional
 - returns the value of the expression $expr$ to the caller

Functions



The screenshot shows a Python 3.4.3 Shell window. It displays the following code:

```
Python 3.4.3 (default, Sep 14 2016, 12:36:27)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> def double(x):
...     return x+x

>>> double(7)
14
>>> double(12)
24
>>>
>>> def sum_list(num_list):
...     sum = 0
...     for i in range(len(num_list)):
...         sum += num_list[i]
...     return sum

>>> sum_list([1,2,3,4])
10
>>> |
```

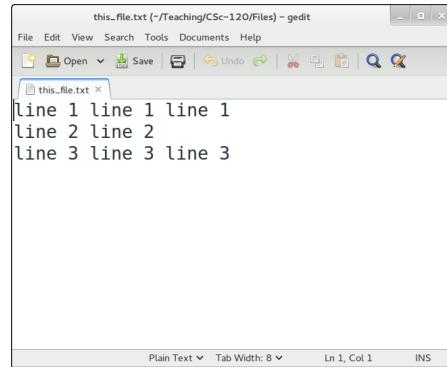
Red arrows point from the text descriptions below to the corresponding parts of the code in the shell window.

- **def *fn_name* (*arg₁*, ..., *arg_n*)**
 - defines a function *fn_name* with *n* arguments *arg₁*, ..., *arg_n*
- **return *expr***
 - optional
 - returns the value of the expression *expr* to the caller
- ***fn_name*(*expr₁*, ..., *expr_n*)**
 - calls *fn_name* with arguments *expr₁*, ..., *expr_n*

python review: reading user input II: file I/O

Reading user input II: file I/O

suppose we want to read
(and process) a file
"this_file.txt"



Reading user input II: file I/O

The diagram illustrates the process of reading a file. On the left, a screenshot of a Python 3.4.3 Shell shows a script being run. A red bracket groups the code for opening the file and the for loop. On the right, a screenshot of a gedit text editor shows the contents of 'this_file.txt' with three lines of text: 'line 1 line 1 line 1', 'line 2 line 2', and 'line 3 line 3 line 3'. Another red bracket groups these three lines of text.

```
Python 3.4.3 (default, Sep 14 2016, 12:36:27)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> infile = open('this_file.txt')
>>> for line in infile:
...     print('"' + line + '"')

"line 1 line 1 line 1
"
"line 2 line 2
"
"line 3 line 3 line 3
"
>>> |
```

this_file.txt (-/Teaching/CSc-120/Files) - gedit

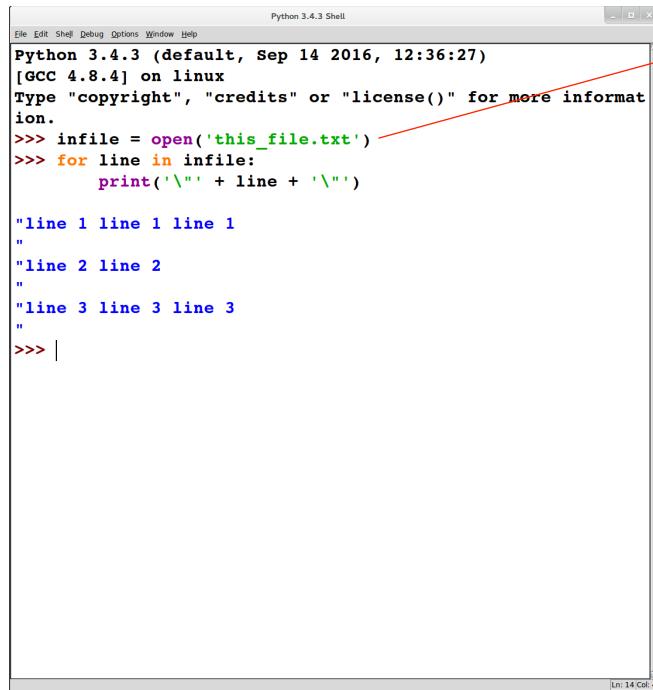
File Edit View Search Tools Documents Help

Plain Text Tab Width: 8 Ln 1, Col 1 INS

line 1 line 1 line 1
line 2 line 2
line 3 line 3 line 3

- open() the file
- read and process the file
- close() the file

Reading user input II: file I/O



The screenshot shows a Python 3.4.3 Shell window. The code reads from a file named 'this_file.txt' and prints its contents. The output shows three lines of text: 'line 1 line 1 line 1', 'line 2 line 2', and 'line 3 line 3 line 3'. A red arrow points from the first bullet point to the 'open' call in the code.

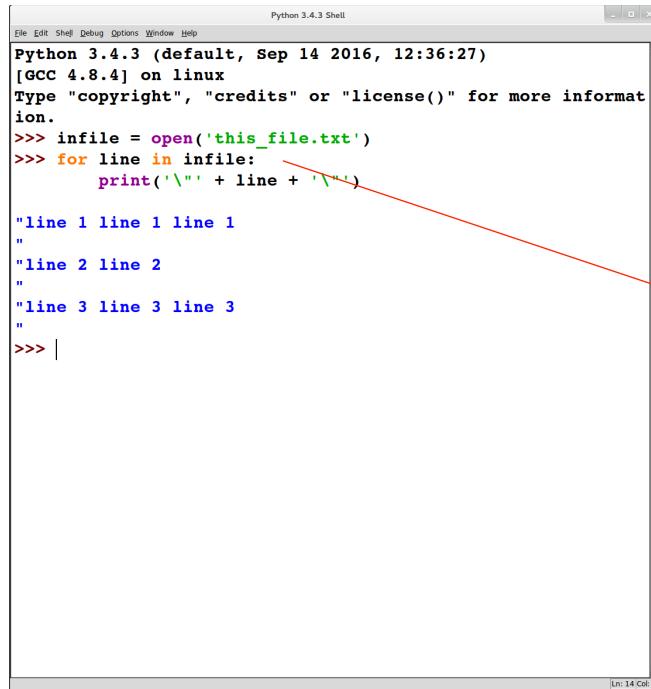
```
Python 3.4.3 (default, Sep 14 2016, 12:36:27)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> infile = open('this_file.txt')
>>> for line in infile:
...     print('"' + line + '"')

"line 1 line 1 line 1
"
"line 2 line 2
"
"line 3 line 3 line 3
"
>>> |
```

- ***fileobj = open(filename)***
 - *filename*: a string
 - *fileobj*: a file object

Reading user input II: file I/O



A screenshot of the Python 3.4.3 Shell window. The title bar says "Python 3.4.3 Shell". The menu bar includes File, Edit, Shelf, Debug, Options, Window, Help. The main area shows Python code and its output:

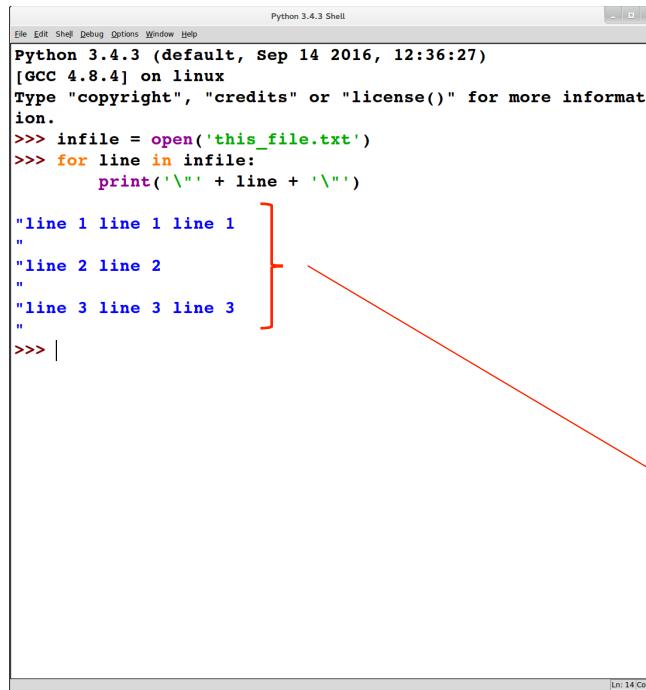
```
Python 3.4.3 (default, Sep 14 2016, 12:36:27)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> infile = open('this_file.txt')
>>> for line in infile:
...     print("'" + line + "'")
...
"line 1 line 1 line 1
"
"line 2 line 2
"
"line 3 line 3 line 3
"
>>> |
```

The code reads the file "this_file.txt" and prints each line enclosed in single quotes. Red arrows point from the explanatory text below to the `open()` call and the `for` loop.

- ***fileobj = open(filename)***
 - *filename*: a string
 - *fileobj*: a file object
- ***for var in fileobj:***
 - reads the file a line at a time
 - assigns the line (a string) to *var*

Reading user input II: file I/O



A screenshot of the Python 3.4.3 Shell window. The title bar says "Python 3.4.3 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, Help. The main window shows Python code and its output. The code reads lines from a file named 'this_file.txt' and prints them. A red bracket on the left side of the code groups the three lines of text being read. A red arrow points from this bracket to the first bullet point in the list below.

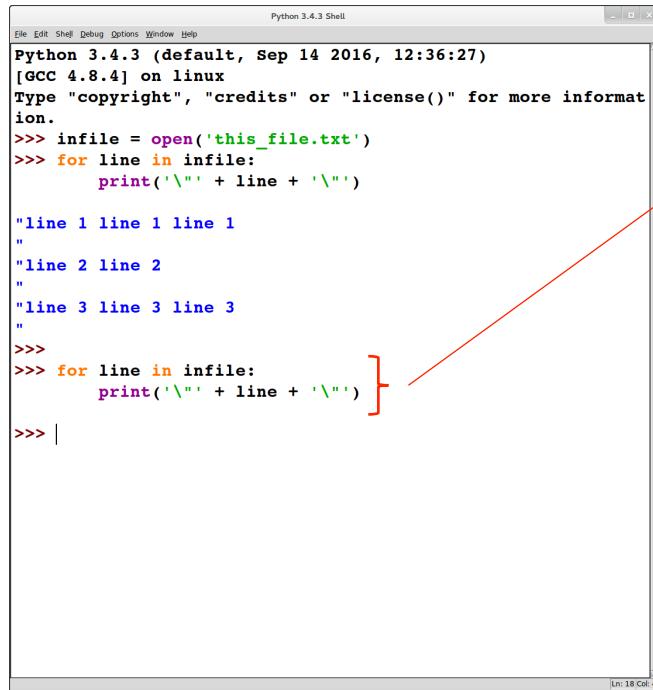
```
Python 3.4.3 (default, Sep 14 2016, 12:36:27)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> infile = open('this_file.txt')
>>> for line in infile:
...     print("'" + line + "'")

"line 1 line 1 line 1"
"
"line 2 line 2"
"
"line 3 line 3 line 3"
"
>>> |
```

- ***fileobj = open(filename)***
 - *filename*: a string
 - *fileobj*: a file object
- ***for var in fileobj:***
 - reads the file a line at a time
 - assigns the line (a string) to *var*
- Note that each line read ends in a newline ('\n') character

Reading user input II: file I/O



A screenshot of the Python 3.4.3 Shell window. The title bar says "Python 3.4.3 Shell". The menu bar includes "File", "Edit", "Shell", "Debug", "Options", "Window", and "Help". The main area displays the following Python code:

```
Python 3.4.3 (default, Sep 14 2016, 12:36:27)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

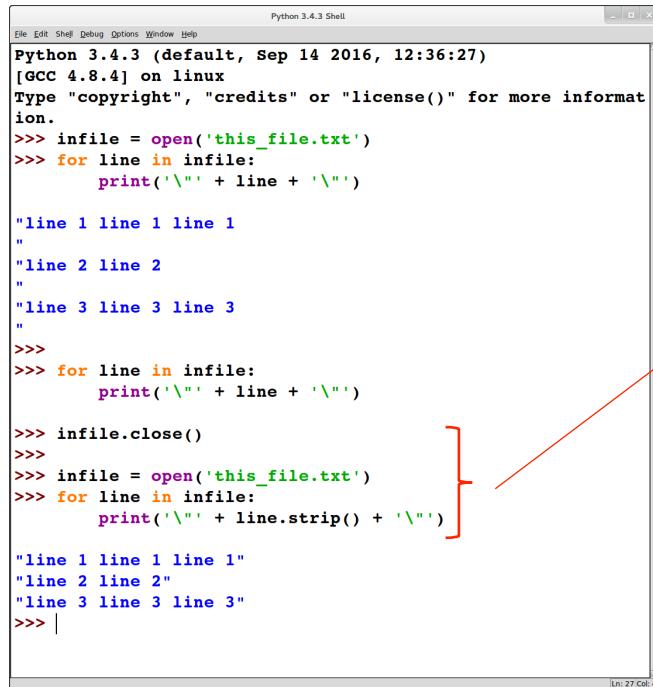
>>> infile = open('this_file.txt')
>>> for line in infile:
...     print('"' + line + '"')

"line 1 line 1 line 1
"
"line 2 line 2
"
"line 3 line 3 line 3
"
>>>
>>> for line in infile:
...     print('"' + line + '"') ]
```

The code reads three lines from a file named "this_file.txt" and prints them back out. A red bracket is placed around the closing brace of the inner loop, indicating that the loop has completed. The status bar at the bottom right shows "Ln: 18 Col: 4".

at this point we've reached the end of the file so there's nothing left to read

Reading user input II: file I/O



```
Python 3.4.3 (default, Sep 14 2016, 12:36:27)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> infile = open('this_file.txt')
>>> for line in infile:
...     print('"' + line + '"')

"line 1 line 1 line 1"
"
"line 2 line 2"
"
"line 3 line 3 line 3"
"
>>>
>>> for line in infile:
...     print('"' + line + '"')

>>> infile.close()
>>>
>>> infile = open('this_file.txt')
>>> for line in infile:
...     print('"' + line.strip() + '"')

"line 1 line 1 line 1"
"line 2 line 2"
"line 3 line 3 line 3"
>>> |
```

at this point we've reached the end of the file so there's nothing left to read

to re-read the file, we have to close it and then re-open it

Reading user input II: file I/O

```
Python 3.4.3 (default, Sep 14 2016, 12:36:27)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> infile = open('this_file.txt')
>>> for line in infile:
...     print('"' + line + '"')

"line 1 line 1 line 1"
"
"line 2 line 2"
"
"line 3 line 3 line 3"
"
>>>
>>> for line in infile:
...     print('"' + line + '"')

>>> infile.close()
>>>
>>> infile = open('this_file.txt')
>>> for line in infile:
...     print('"' + line.strip() + '"')

"line 1 line 1 line 1"
"line 2 line 2"
"line 3 line 3 line 3"
>>> |
```

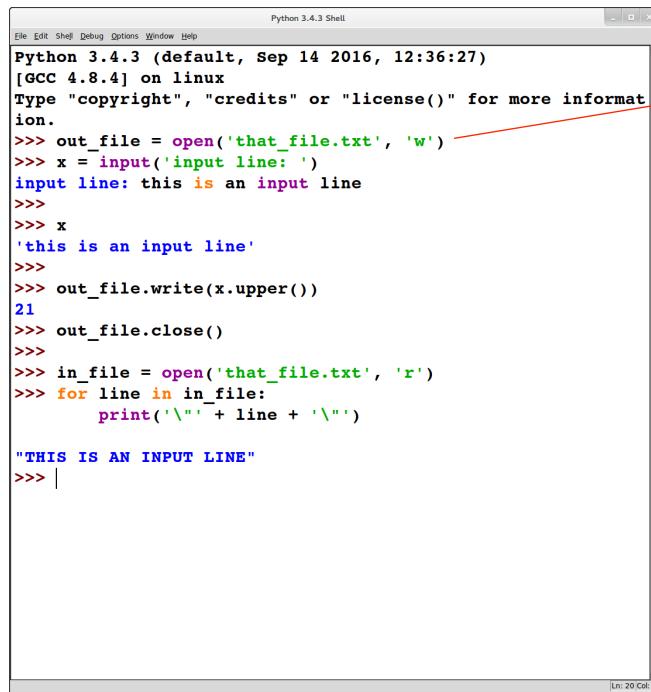
The screenshot shows a Python 3.4.3 Shell window. The code reads the contents of 'this_file.txt' and prints each line with quotes around them. It then closes the file and re-opens it to read the same lines again, this time using `line.strip()` to remove the trailing newlines before printing.

at this point we've reached the end of the file so there's nothing left to read

to re-read the file, we have to close it and then re-open it

NOTE: we can use `strip()` to get rid of the newline character at the end of each line

Writing output to a file

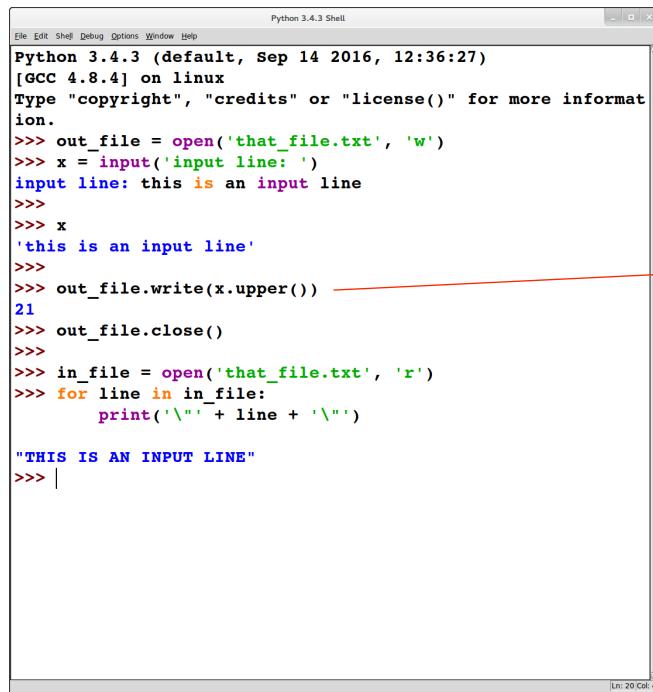


Python 3.4.3 (default, Sep 14 2016, 12:36:27)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.
>>> out_file = open('that_file.txt', 'w')
>>> x = input('input line: ')
input line: this is an input line
>>>
>>> x
'this is an input line'
>>>
>>> out_file.write(x.upper())
21
>>> out_file.close()
>>>
>>> in_file = open('that_file.txt', 'r')
>>> for line in in_file:
 print('"' + line + '"')

"THIS IS AN INPUT LINE"
>>> |

open(filename, "w") : opens *filename* in write mode, i.e., for output

Writing output to a file



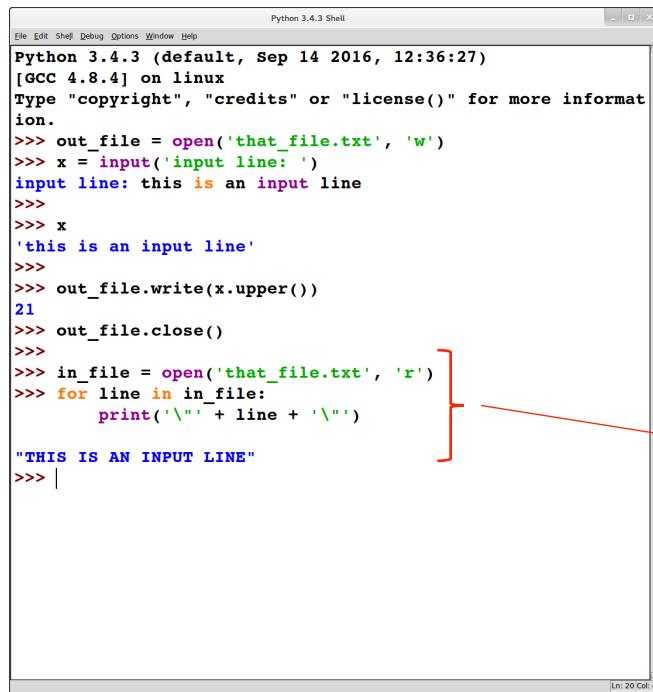
Python 3.4.3 (default, Sep 14 2016, 12:36:27)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.
>>> out_file = open('that_file.txt', 'w')
>>> x = input('input line: ')
input line: this is an input line
>>>
>>> x
'this is an input line'
>>>
>>> out_file.write(x.upper())
21
>>> out_file.close()
>>>
>>> in_file = open('that_file.txt', 'r')
>>> for line in in_file:
 print('"' + line + '"')

"THIS IS AN INPUT LINE"
>>> |

open(filename, "w") : opens *filename* in write mode, i.e., for output

fileobj.write(string) : writes *string* to *fileobj*

Writing output to a file



```
Python 3.4.3 (default, Sep 14 2016, 12:36:27)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> out_file = open('that_file.txt', 'w')
>>> x = input('input line: ')
input line: this is an input line
>>>
>>> x
'this is an input line'
>>>
>>> out_file.write(x.upper())
21
>>> out_file.close()
>>>
>>> in_file = open('that_file.txt', 'r')
>>> for line in in_file:
    print('"' + line + '"')

"THIS IS AN INPUT LINE"
>>> |
```

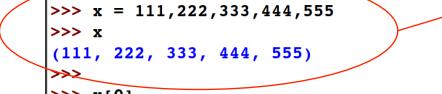
open(filename, "w") : opens *filename* in write mode, i.e., for output

fileobj.write(string) : writes *string* to *fileobj*

open the file in read mode ("r") to see what was written

python review: tuples

Tuples

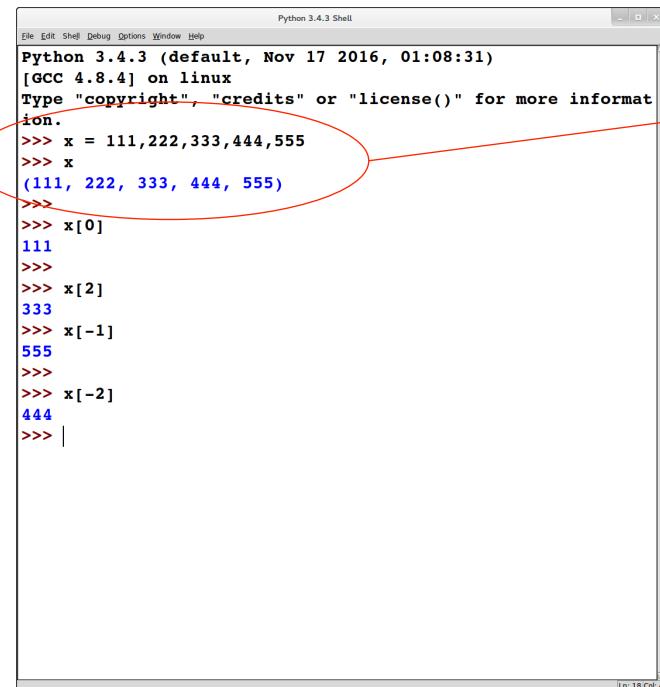


```
Python 3.4.3 (default, Nov 17 2016, 01:08:31)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> x = 111,222,333,444,555
>>> x
(111, 222, 333, 444, 555)
>>>
>>> x[0]
111
>>>
>>> x[2]
333
>>> x[-1]
555
>>>
>>> x[-2]
444
>>> |
```

a tuple is a sequence of values (like lists)

Tuples



A screenshot of the Python 3.4.3 Shell window. The title bar says "Python 3.4.3 Shell". The menu bar includes File, Edit, Shelf, Debug, Options, Window, and Help. The main window shows the following Python session:

```
Python 3.4.3 (default, Nov 17 2016, 01:08:31)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> x = 111,222,333,444,555
>>> x
(111, 222, 333, 444, 555)
>>>
>>> x[0]
111
>>>
>>> x[2]
333
>>> x[-1]
555
>>>
>>> x[-2]
444
>>> |
```

The line "x = 111,222,333,444,555" is circled with a red oval.

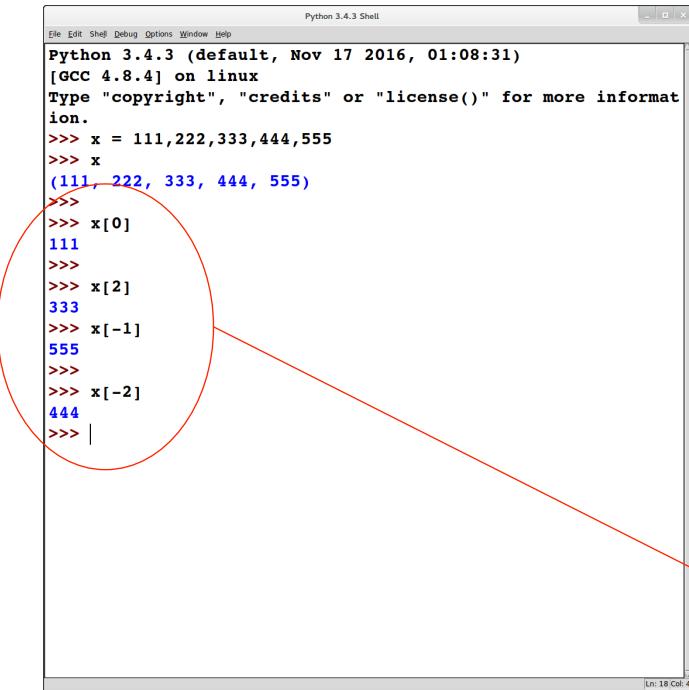
a tuple is a sequence of values (like lists)

tuples use parens ()

- by contrast, lists use square brackets []
 - parens can be omitted if no confusion is possible
- special cases for tuples:
 - empty tuple: ()
 - single-element tuple: must have comma after the element:

(111,)

Tuples



A screenshot of the Python 3.4.3 Shell window. The menu bar includes File, Edit, Shelf, Debug, Options, Window, and Help. The title bar says "Python 3.4.3 Shell". The main area shows the following Python session:

```
Python 3.4.3 (default, Nov 17 2016, 01:08:31)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> x = 111,222,333,444,555
>>> x
(111, 222, 333, 444, 555)
>>> x[0]
111
>>>
>>> x[2]
333
>>> x[-1]
555
>>>
>>> x[-2]
444
>>> |
```

A red circle highlights the assignment statement `x = 111,222,333,444,555` and the resulting tuple value `(111, 222, 333, 444, 555)`. A red line points from this circled area to the text "tuples use parens ()".

a tuple is a sequence of values (like lists)

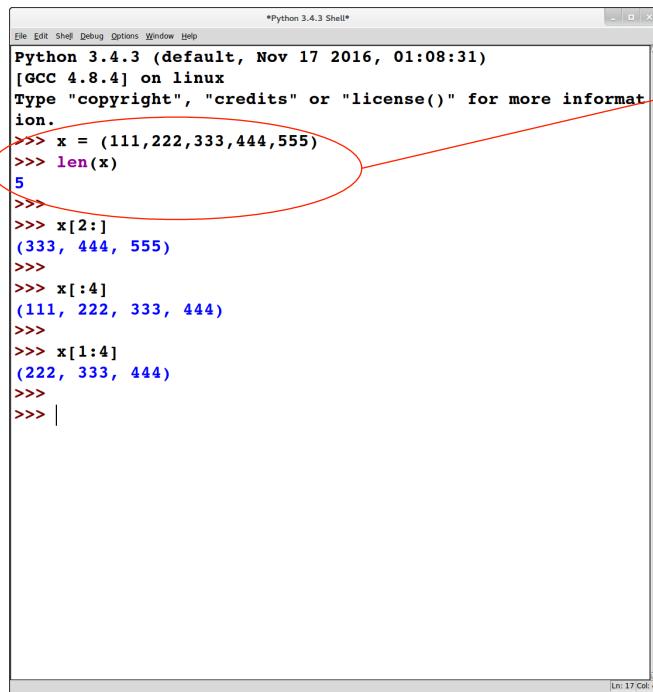
tuples use parens ()

- by contrast, lists use square brackets []
 - parens can be omitted if no confusion is possible
- special cases for tuples:
 - empty tuple: ()
 - single-element tuple: must have comma after the element:

`(111,)`

indexing in tuples works similarly to strings and lists

Tuples



The screenshot shows a Python 3.4.3 Shell window. The code demonstrates various tuple operations:

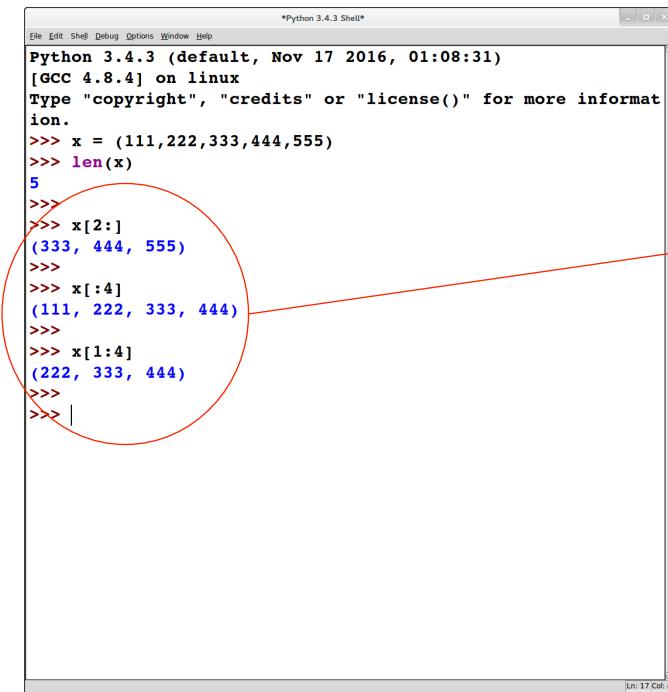
```
*Python 3.4.3 Shell*
File Edit Shelf Debug Options Window Help
Python 3.4.3 (default, Nov 17 2016, 01:08:31)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> x = (111,222,333,444,555)
>>> len(x)
5
>>>
>>> x[2:]
(333, 444, 555)
>>>
>>> x[:4]
(111, 222, 333, 444)
>>>
>>> x[1:4]
(222, 333, 444)
>>>
>>> |
```

A red oval highlights the line `x = (111,222,333,444,555)`, and another red oval highlights the line `len(x)`.

computing a length of a tuple: similar to strings and lists

Tuples



A screenshot of the Python 3.4.3 Shell window. The title bar says "Python 3.4.3 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, Help. The Python version information is displayed: "Python 3.4.3 (default, Nov 17 2016, 01:08:31) [GCC 4.8.4] on linux". A note says "Type 'copyright', 'credits' or 'license()' for more information." The code entered in the shell is:

```
File Edit Shell Debug Options Window Help
*Python 3.4.3 Shell*
Python 3.4.3 (default, Nov 17 2016, 01:08:31)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

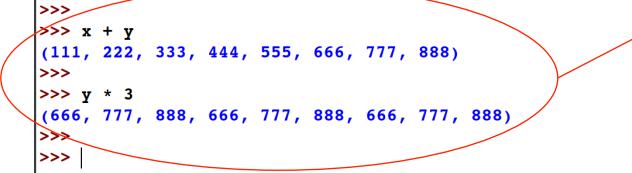
>>> x = (111,222,333,444,555)
>>> len(x)
5
>>> x[2:]
(333, 444, 555)
>>>
>>> x[:4]
(111, 222, 333, 444)
>>>
>>> x[1:4]
(222, 333, 444)
>>>
>>> |
```

The output of the slice operations is highlighted with a red oval.

computing a length of a tuple: similar to strings and lists

computing slices of a tuple: similar to strings and lists

Tuples

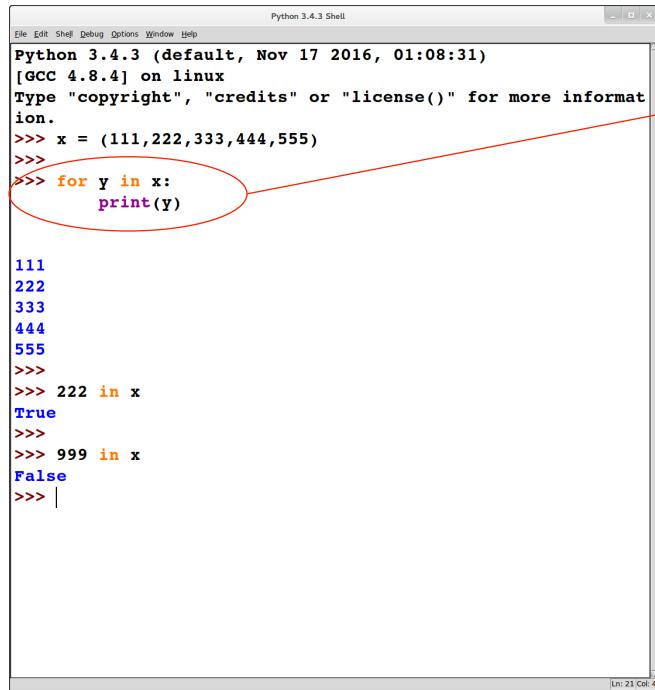


```
Python 3.4.3 (default, Nov 17 2016, 01:08:31)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> x = 111,222,333,444,555
>>> y = (666,777,888)
>>>
>>> x + y
(111, 222, 333, 444, 555, 666, 777, 888)
>>>
>>> y * 3
(666, 777, 888, 666, 777, 888, 666, 777, 888)
>>>
>>> |
```

+ and * work similarly on tuples as for lists and strings

Tuples



A screenshot of the Python 3.4.3 Shell window. The title bar says "Python 3.4.3 Shell". The menu bar includes File, Edit, Shelf, Debug, Options, Window, Help. The main area shows Python code and its output:

```
Python 3.4.3 (default, Nov 17 2016, 01:08:31)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

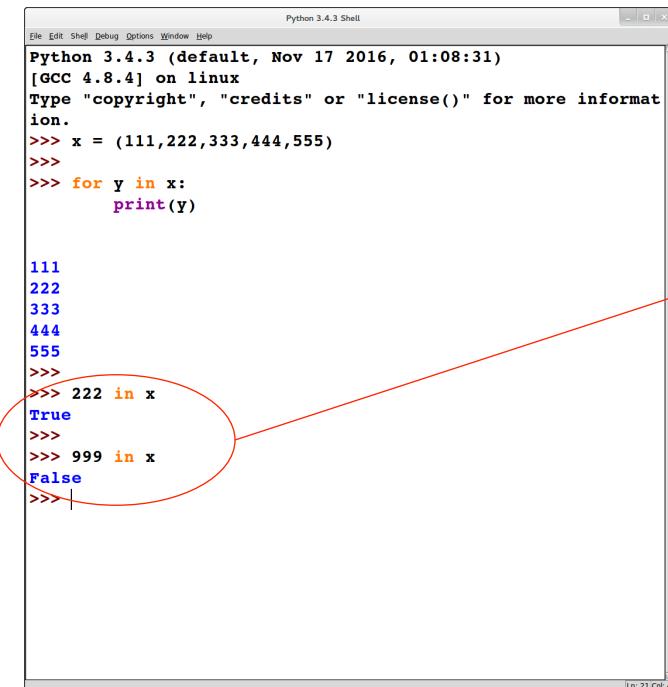
>>> x = (111,222,333,444,555)
>>>
>>> for y in x:
    print(y)

111
222
333
444
555
>>>
>>> 222 in x
True
>>>
>>> 999 in x
False
>>> |
```

The line `for y in x:` is highlighted with a red oval.

iterating through the elements of a tuple: similar to lists and strings

Tuples



A screenshot of the Python 3.4.3 Shell window. The menu bar includes File, Edit, Shelf, Debug, Options, Window, Help. The title bar says "Python 3.4.3 Shell". The code area shows:

```
Python 3.4.3 (default, Nov 17 2016, 01:08:31)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> x = (111,222,333,444,555)
>>>
>>> for y in x:
    print(y)

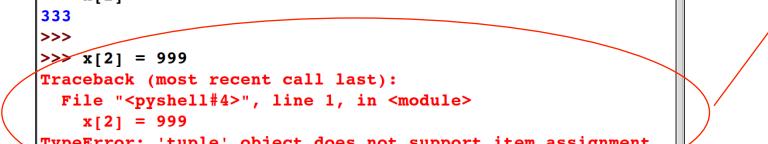
111
222
333
444
555
>>>
>>> 222 in x
True
>>>
>>> 999 in x
False
>>>
```

The line `222 in x` is highlighted with a red oval.

iterating through the elements of a tuple: similar to lists and strings

checking membership in a tuple: similar to lists and strings

Tuples



```
Python 3.4.3 (default, Nov 17 2016, 01:08:31)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.
>>> x = (111,222,333,444,555)
>>>
>>> x[2]
333
>>>
>>> x[2] = 999
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    x[2] = 999
TypeError: 'tuple' object does not support item assignment
>>> |
```

tuples are not mutable

Sequence types: mutability

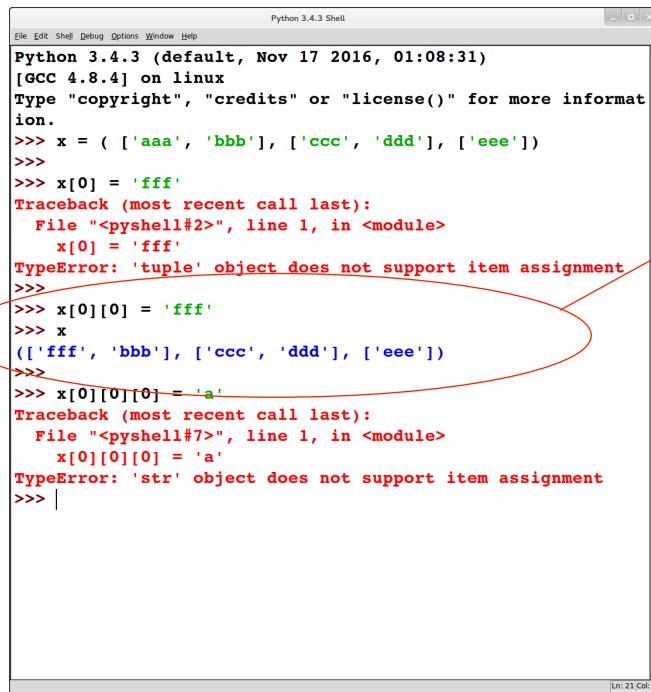
```
Python 3.4.3 (default, Nov 17 2016, 01:08:31)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> x = ( ['aaa', 'bbb'], ['ccc', 'ddd'], ['eee'])

>>> x[0] = 'fff'
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    x[0] = 'fff'
TypeError: 'tuple' object does not support item assignment
>>>
>>> x[0][0] = 'fff'
>>> x
(['fff', 'bbb'], ['ccc', 'ddd'], ['eee'])
>>>
>>> x[0][0][0] = 'a'
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    x[0][0][0] = 'a'
TypeError: 'str' object does not support item assignment
>>> |
```

tuples are immutable

Sequence types: mutability



A screenshot of the Python 3.4.3 Shell window. The title bar says "Python 3.4.3 Shell". The code entered is:

```
Python 3.4.3 (default, Nov 17 2016, 01:08:31)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> x = ( ['aaa', 'bbb'], ['ccc', 'ddd'], ['eee'])
>>>
>>> x[0] = 'fff'
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    x[0] = 'fff'
TypeError: 'tuple' object does not support item assignment
>>>
>>> x[0][0] = 'fff'
>>> x
(['fff', 'bbb'], ['ccc', 'ddd'], ['eee'])
>>>
>>> x[0][0][0] = 'a'
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    x[0][0][0] = 'a'
TypeError: 'str' object does not support item assignment
>>> |
```

The line `x[0] = 'fff'` is circled in red. A red arrow points from this circled line to the text "tuples are immutable" below.

tuples are immutable

lists are mutable (even if the list is an element of a [immutable] tuple)

Sequence types: mutability

```
Python 3.4.3 (default, Nov 17 2016, 01:08:31)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> x = ( ['aaa', 'bbb'], ['ccc', 'ddd'], ['eee'])
>>>
>>> x[0] = 'fff'
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    x[0] = 'fff'
TypeError: 'tuple' object does not support item assignment
>>>
>>> x[0][0] = 'fff'
>>> x
(['fff', 'bbb'], ['ccc', 'ddd'], ['eee'])
>>>
>>> x[0][0][0] = 'a'
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    x[0][0][0] = 'a'
TypeError: 'str' object does not support item assignment
>>> |
```

tuples are immutable

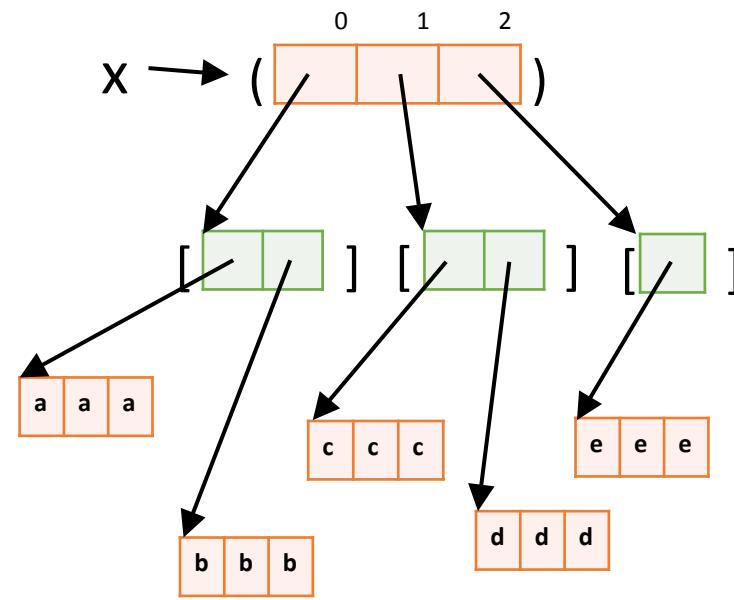
lists are mutable (even if the list is an element of a [immutable] tuple)

strings are immutable (even if the string is an element of a [mutable] list)

Sequence types: mutability

```
Python 3.4.3 (default, Nov 17 2016, 01:08:31)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> x = ( ['aaa', 'bbb'], ['ccc', 'ddd'], ['eee'])
>>>
>>> x[0] = 'fff'
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    x[0] = 'fff'
TypeError: 'tuple' object does not support item assignment
>>>
>>> x[0][0] = 'fff'
>>> x
(['fff', 'bbb'], ['ccc', 'ddd'], ['eee'])
>>>
>>> x[0][0][0] = 'a'
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    x[0][0][0] = 'a'
TypeError: 'str' object does not support item assignment
>>> |
```



tuple
(immutable)

list
(mutable)

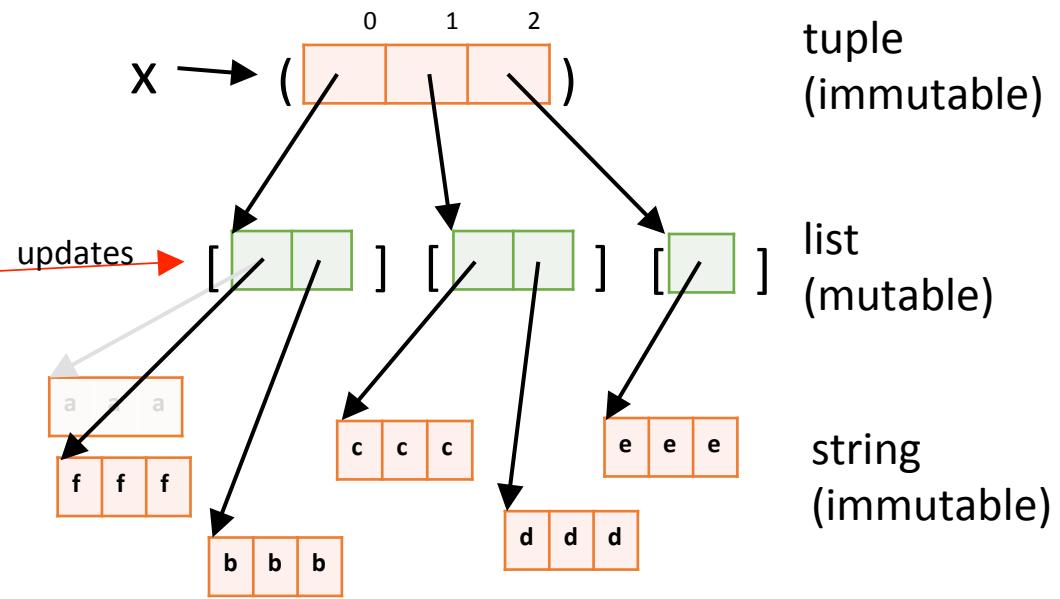
string
(immutable)

Sequence types: mutability

```
Python 3.4.3 (default, Nov 17 2016, 01:08:31)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> x = ( ['aaa', 'bbb'], ['ccc', 'ddd'], ['eee'])

>>> x[0] = 'fff'
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    x[0] = 'fff'
TypeError: 'tuple' object does not support item assignment
>>> x[0][0] = 'fff'
('fff', 'bbb'], ['ccc', 'ddd'], ['eee'])
>>>
>>> x[0][0][0] = 'a'
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    x[0][0][0] = 'a'
TypeError: 'str' object does not support item assignment
>>> |
```



tuple
(immutable)

list
(mutable)

string
(immutable)

Why use tuples?

At the implementation level, tuples are much simpler than lists:

- lists are mutable; tuples are immutable
 - this means that the implementation can process tuples without having to worry about the possibility of updates
- lists have methods (e.g., append); tuples do not have methods

⇒ Tuples can be implemented more efficiently than lists

Summary: sequence types

Sequence types include: strings, lists, and tuples

Operation	Result
<code>x in s</code>	True if an item of <code>s</code> is equal to <code>x</code> , else False
<code>x not in s</code>	False if an item of <code>s</code> is equal to <code>x</code> , else True
<code>s + t</code>	the concatenation of <code>s</code> and <code>t</code>
<code>s * n</code> or <code>n * s</code>	equivalent to adding <code>s</code> to itself <code>n</code> times
<code>s[i]</code>	<code>i</code> th item of <code>s</code> , origin 0
<code>s[i:j]</code>	slice of <code>s</code> from <code>i</code> to <code>j</code>
<code>s[i:j:k]</code>	slice of <code>s</code> from <code>i</code> to <code>j</code> with step <code>k</code>
<code>len(s)</code>	length of <code>s</code>
<code>min(s)</code>	smallest item of <code>s</code>
<code>max(s)</code>	largest item of <code>s</code>
<code>s.index(x[, i[, j]])</code>	index of the first occurrence of <code>x</code> in <code>s</code> (at or after index <code>i</code> and before index <code>j</code>)
<code>s.count(x)</code>	total number of occurrences of <code>x</code> in <code>s</code>

The elements
are: $i, i+k, i+2k, \dots$

Source: <https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range>

EXERCISE

```
>>> x = [ (1, 2, 3), (4, 5, 6), (7, 8, 9) ]
```

```
>>> x[0][0] = (2, 3, 4)
```



*what do you think will be
printed out?*

```
>>> x[0] = [ 2, 3, 4 ]
```



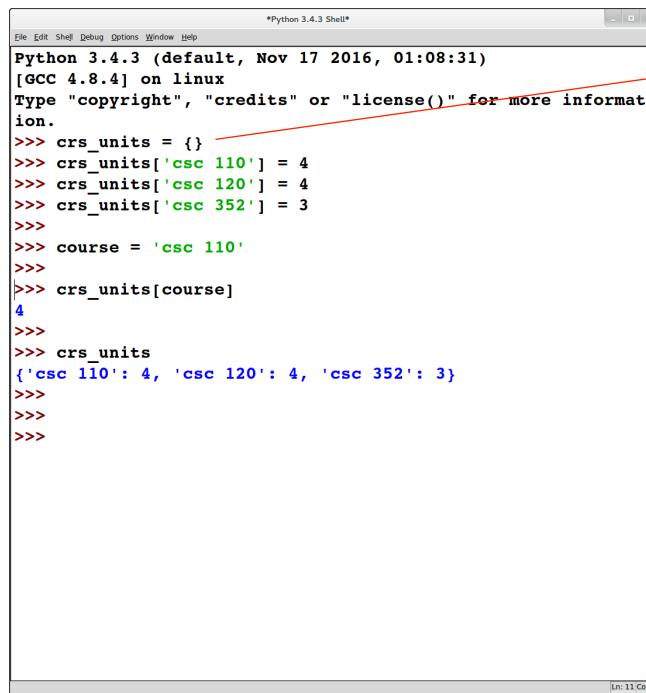
*what do you think will be
printed out?*

python review: dictionaries

Dictionaries

- A dictionary is like an array, but it can be indexed using strings (or numbers, or tuples, or any immutable type)
 - the values used as indexes for a particular dictionary are called its *keys*
 - think of a dictionary as an unordered collection of *key : value* pairs
 - empty dictionary: {}
- It is an error to index into a dictionary using a non-existent key

Dictionaries



The screenshot shows a Python 3.4.3 Shell window. The code demonstrates how to create an empty dictionary and then add key-value pairs to it.

```
*Python 3.4.3 Shell*
File Edit Shelf Debug Options Window Help
Python 3.4.3 (default, Nov 17 2016, 01:08:31)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

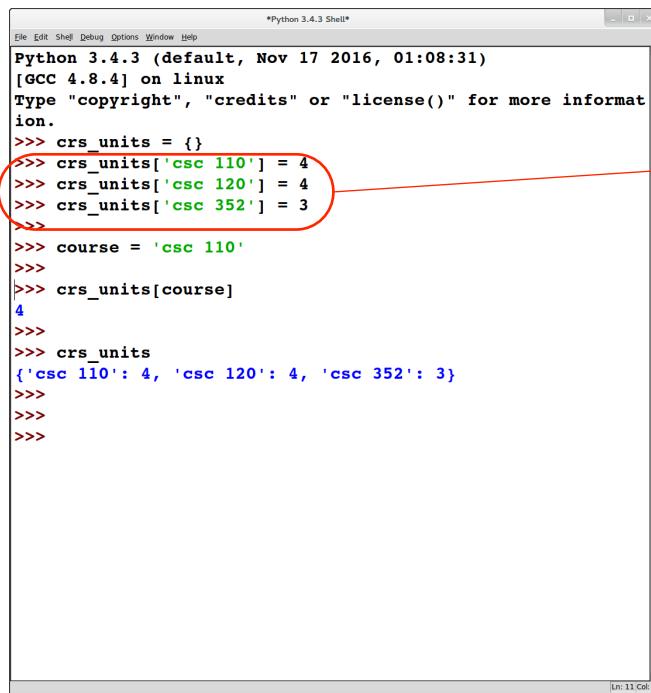
>>> crs_units = {}
>>> crs_units['csc 110'] = 4
>>> crs_units['csc 120'] = 4
>>> crs_units['csc 352'] = 3
>>>
>>> course = 'csc 110'
>>>
>>> crs_units[course]
4
>>>
>>> crs_units
{'csc 110': 4, 'csc 120': 4, 'csc 352': 3}
>>>
>>>
```

A red line points from the text "empty dictionary" to the line where the dictionary is first created: `>>> crs_units = {}`.

A red line also points from the text "empty dictionary" to the line where the dictionary is printed: `>>> crs_units`.

empty dictionary

Dictionaries



The screenshot shows a Python 3.4.3 Shell window. The code demonstrates how to create an empty dictionary and then add items to it:

```
*Python 3.4.3 Shell*
File Edit Shelf Debug Options Window Help
Python 3.4.3 (default, Nov 17 2016, 01:08:31)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

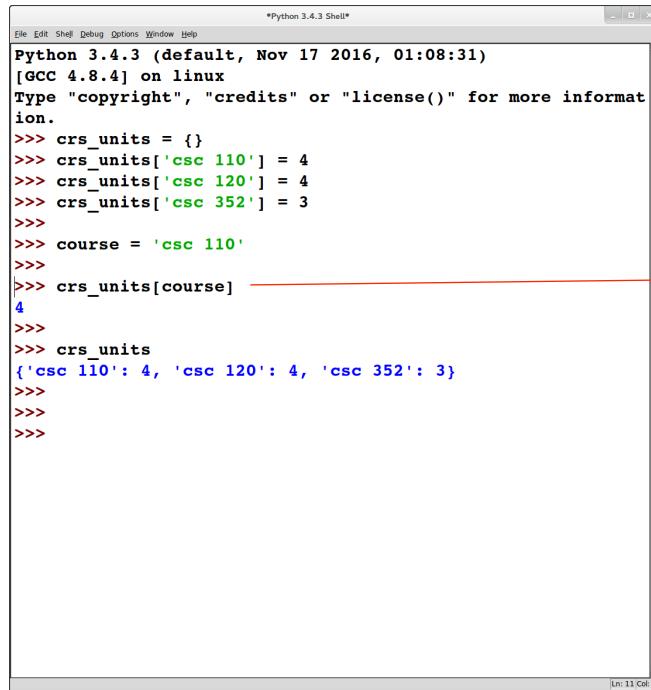
>>> crs_units = {}
>>> crs_units['csc 110'] = 4
>>> crs_units['csc 120'] = 4
>>> crs_units['csc 352'] = 3
>>>
>>> course = 'csc 110'
>>>
>>> crs_units[course]
4
>>>
>>> crs_units
{'csc 110': 4, 'csc 120': 4, 'csc 352': 3}
>>>
>>>
```

empty dictionary

populating the dictionary

- in this example, one item at a time

Dictionaries



The screenshot shows a Python 3.4.3 Shell window with the title "*Python 3.4.3 Shell*". The window displays the following code and its output:

```
File Edit Shelf Debug Options Window Help
Python 3.4.3 (default, Nov 17 2016, 01:08:31)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> crs_units = {}
>>> crs_units['csc 110'] = 4
>>> crs_units['csc 120'] = 4
>>> crs_units['csc 352'] = 3
>>>
>>> course = 'csc 110'
>>>
>>> crs_units[course]
4
>>>
>>> crs_units
{'csc 110': 4, 'csc 120': 4, 'csc 352': 3}
>>>
>>>
```

The code defines an empty dictionary `crs_units`, adds three key-value pairs, and then retrieves the value for the key 'csc 110'.

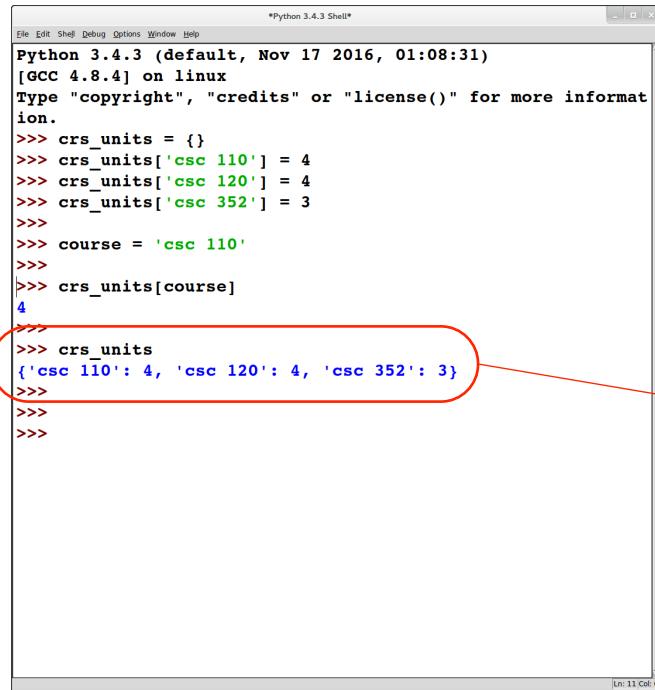
empty dictionary

populating the dictionary

- in this example, one item at a time

looking up the dictionary (indexing)

Dictionaries



The screenshot shows a Python 3.4.3 Shell window. The code demonstrates creating an empty dictionary, adding items to it, and then looking up an item. A red oval highlights the output of the print statement, which shows the dictionary with three key-value pairs.

```
*Python 3.4.3 Shell*
File Edit Shell Debug Options Window Help
Python 3.4.3 (default, Nov 17 2016, 01:08:31)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> crs_units = {}
>>> crs_units['csc 110'] = 4
>>> crs_units['csc 120'] = 4
>>> crs_units['csc 352'] = 3
>>>
>>> course = 'csc 110'
>>>
>>> crs_units[course]
4
>>>
>>> crs_units
{'csc 110': 4, 'csc 120': 4, 'csc 352': 3}
>>>
>>>
```

empty dictionary

populating the dictionary

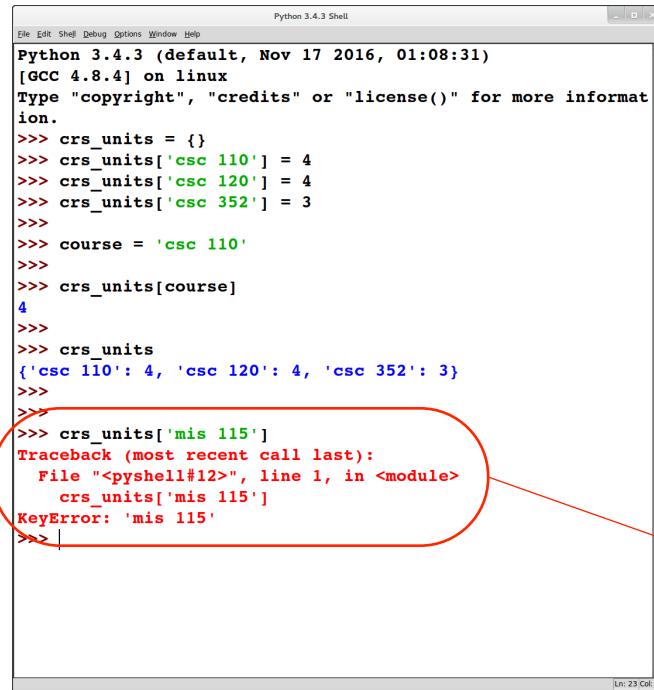
- in this example, one item at a time

looking up the dictionary (indexing)

looking at the dictionary

- we can use this syntax to populate the dictionary too

Dictionaries



The screenshot shows a Python 3.4.3 Shell window. The code demonstrates creating an empty dictionary, adding key-value pairs, and then attempting to access a key that does not exist, which results in a `KeyError`.

```
Python 3.4.3 (default, Nov 17 2016, 01:08:31)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> crs_units = {}
>>> crs_units['csc 110'] = 4
>>> crs_units['csc 120'] = 4
>>> crs_units['csc 352'] = 3
>>>
>>> course = 'csc 110'
>>>
>>> crs_units[course]
4
>>>
>>> crs_units
{'csc 110': 4, 'csc 120': 4, 'csc 352': 3}
>>>
>>> crs_units['mis 115']
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    crs_units['mis 115']
KeyError: 'mis 115'
>>>
```

empty dictionary

populating the dictionary

- in this example, one item at a time

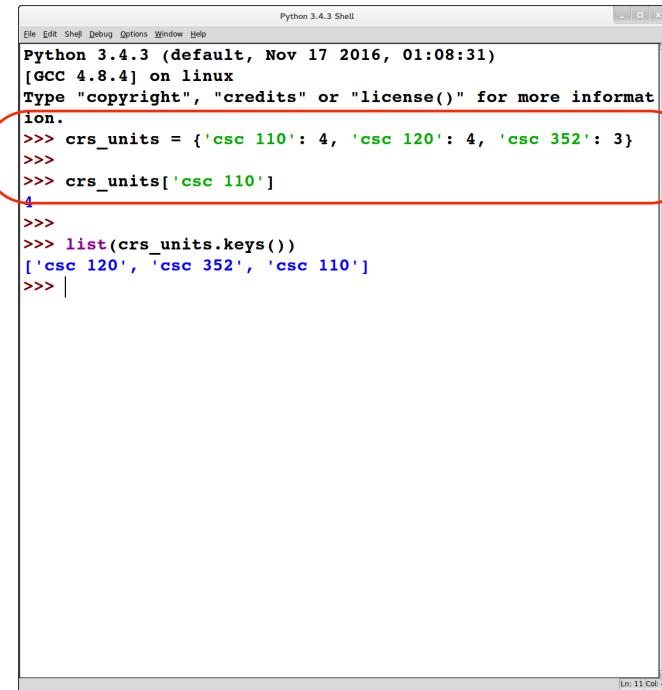
looking up the dictionary (indexing)

looking at the dictionary

- we can use this syntax to populate the dictionary too

indexing with a key not in the dictionary is an error (`KeyError`)

Dictionaries



A screenshot of the Python 3.4.3 Shell window. The menu bar includes File, Edit, Shelf, Debug, Options, Window, and Help. The title bar says "Python 3.4.3 Shell". The code area shows:

```
Python 3.4.3 (default, Nov 17 2016, 01:08:31)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

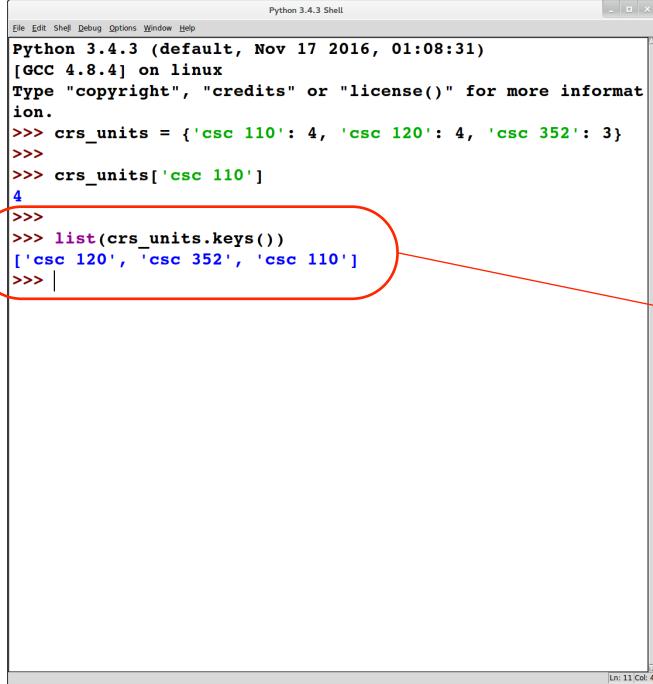
>>> crs_units = {'csc 110': 4, 'csc 120': 4, 'csc 352': 3}
>>>
>>> crs_units['csc 110']
4
>>>
>>> list(crs_units.keys())
['csc 120', 'csc 352', 'csc 110']
>>> |
```

The first three lines of code are highlighted with a red circle.

initializing the dictionary

- in this example, several items at once

Dictionaries



The screenshot shows a Python 3.4.3 Shell window. The code demonstrates initializing a dictionary with multiple items and then getting a list of keys:

```
Python 3.4.3 (default, Nov 17 2016, 01:08:31)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> crs_units = {'csc 110': 4, 'csc 120': 4, 'csc 352': 3}
>>>
>>> crs_units['csc 110']
4
>>>
>>> list(crs_units.keys())
['csc 120', 'csc 352', 'csc 110']
>>> |
```

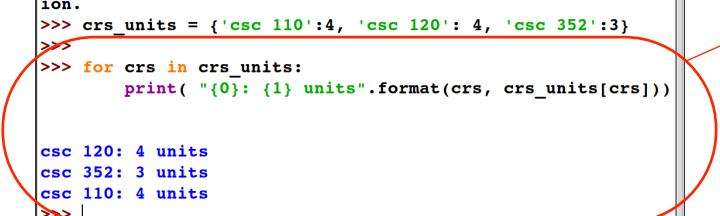
initializing the dictionary

- in this example, several items at once

getting a list of keys in the dictionary

- useful since it's an error to index into a dictionary with a key that is not in it

Dictionaries



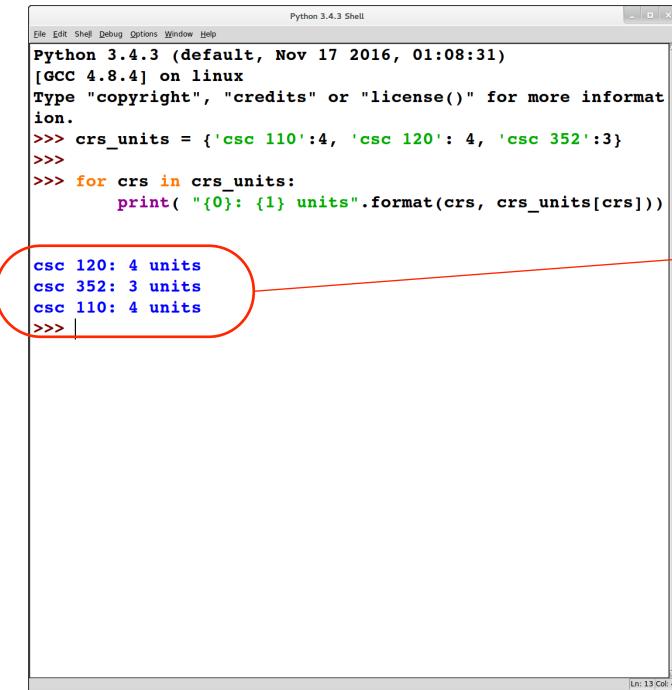
```
Python 3.4.3 (default, Nov 17 2016, 01:08:31)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> crs_units = {'csc 110':4, 'csc 120': 4, 'csc 352':3}
>>>
>>> for crs in crs_units:
    print( "{0}: {1} units".format(crs, crs_units[crs]))

csc 120: 4 units
csc 352: 3 units
csc 110: 4 units
```

We can use a **for** loop to iterate through a dictionary

Dictionaries



```
Python 3.4.3 (default, Nov 17 2016, 01:08:31)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.
>>> crs_units = {'csc 110':4, 'csc 120': 4, 'csc 352':3}
>>>
>>> for crs in crs_units:
    print( "{0}: {1} units".format(crs, crs_units[crs]))
csc 120: 4 units
csc 352: 3 units
csc 110: 4 units
>>>
```

A red oval highlights the output of the print statement, which shows three key-value pairs: 'csc 120: 4 units', 'csc 352: 3 units', and 'csc 110: 4 units'. A red line connects this oval to the explanatory text on the right.

We can use a **for** loop to iterate through a dictionary

Notice that this iteration may not list the items in the dictionary in the same order as when they were inserted

EXERCISE

```
>>> crs_units = { 'csc 352' : 3, 'csc 120': 4, 'csc 110': 4 }  
>>> for crs in    
    print( "{0} : {1} units".format( crs, crs_units[crs] )
```

csc 110 : 4 units
csc 120 : 4 units
csc 352 : 3 units
>>>

*How can we get the dictionary contents to be printed out in sorted order of the keys?
(I.e., what goes in the box?)*