CSc 120
Introduction to Computer Programing II

Python review

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
- 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
>>>
```

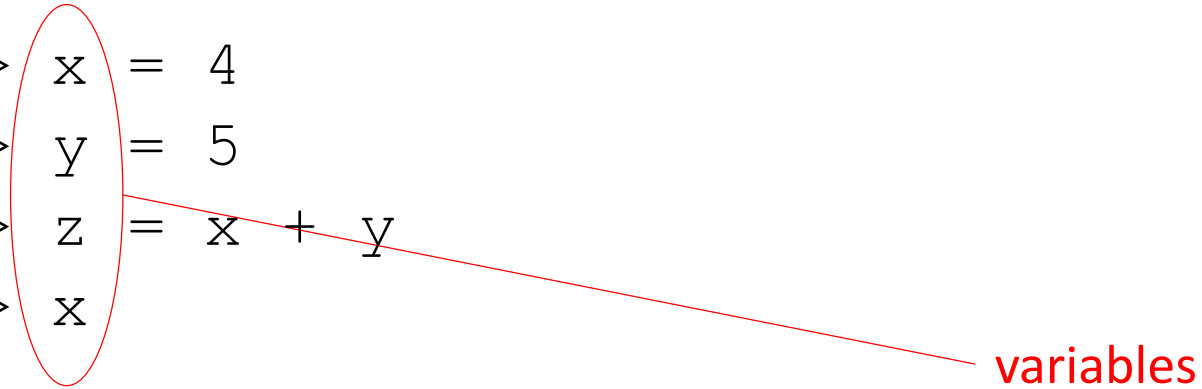
python basics

```
>>> 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

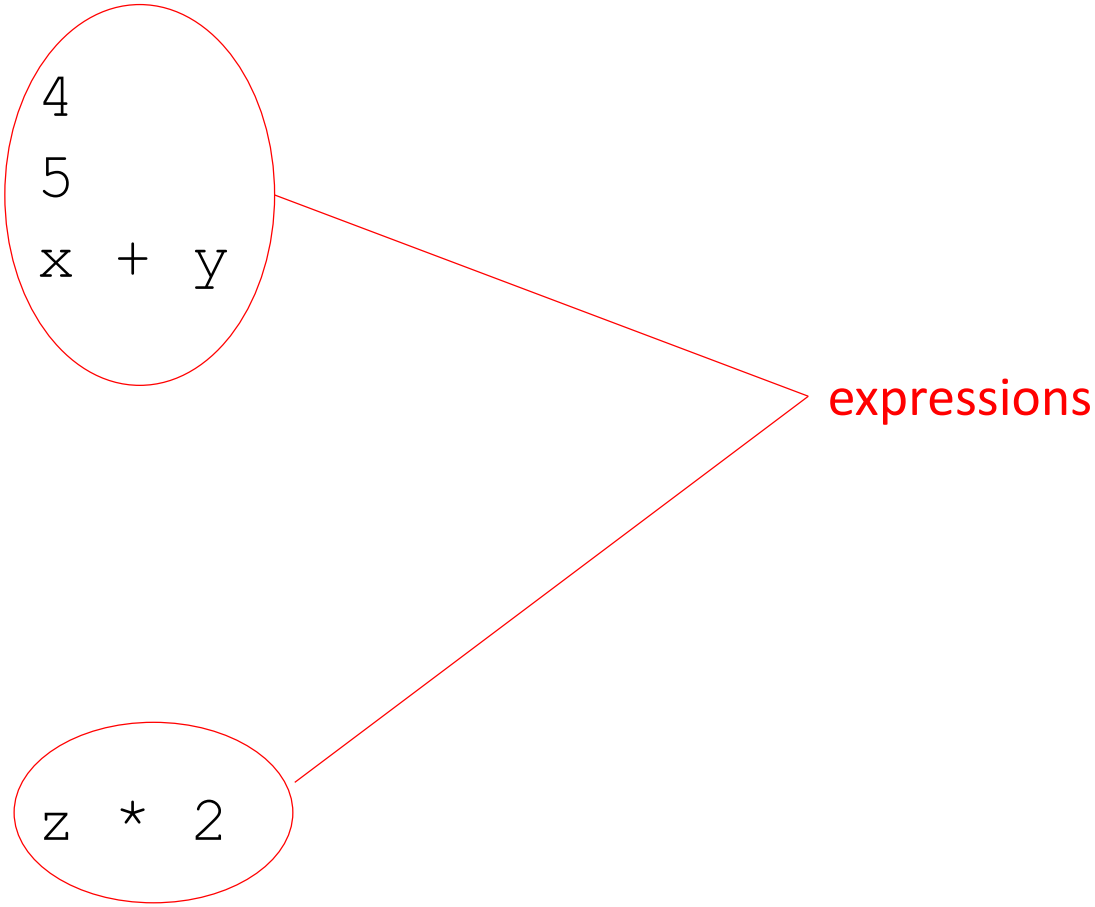
python basics

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



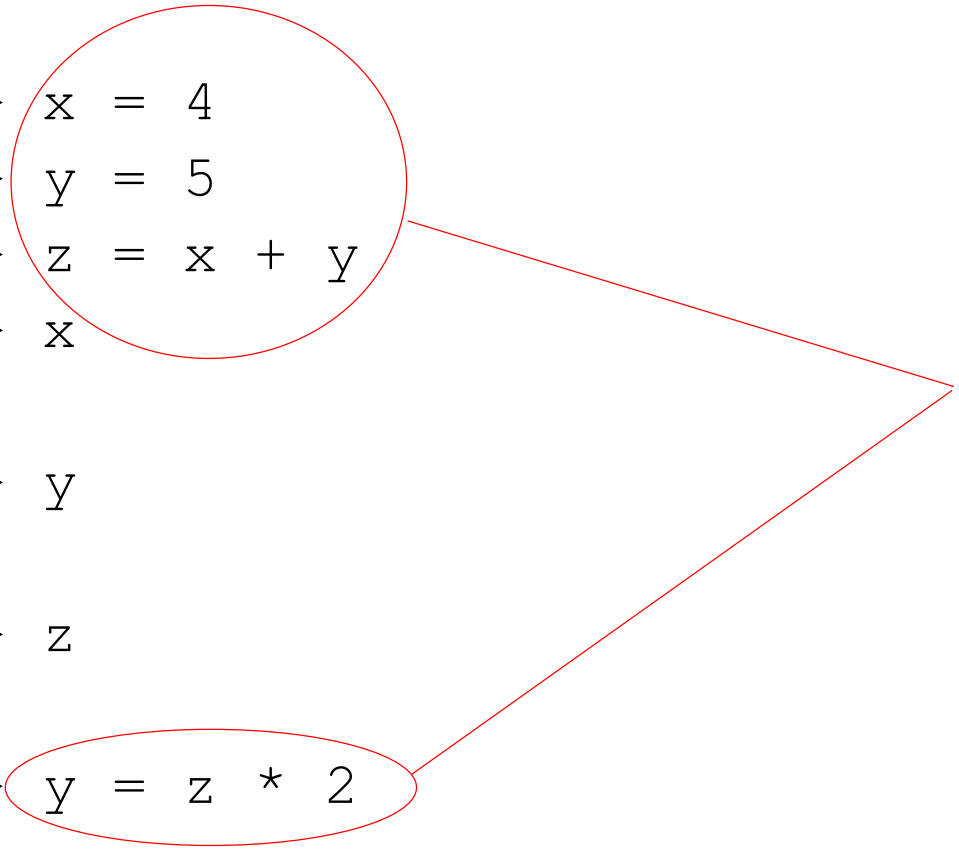
python basics

```
>>> 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
>>>
```



assignment statements

python basics

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

$x_1, x_2, ..., x_n = exp_1, exp_2, ..., exp_n$

Behavior:

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

Comparison and Booleans

```
>>> x, y, z = 11, 22, 33
>>> x
11
>>> y
22
>>> z
33
>>> x < y
True
>>> y == z
False
```

Comparision operations:

<, >, ==, >=, <=, !=

Lower precedence than arithmetic operations.

Yield boolean values:

True

False

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: basics of strings

Basics of strings

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

a string is a sequence of characters
(letters, numbers, and other symbols)

Basics of strings

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

a string is a sequence of characters (letters, numbers and other symbols)

a string literal is enclosed in quotes

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

Basics of strings

```
>>> text = '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 of characters

- we can index into a string to get the characters

Basics of strings

```
>>> text = '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 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 = '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 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

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

len(x) : length of a string x

Basics of strings

```
>>> x = 'abcDE_fgHIJ_01234'
>>> x
'abcDE_fgHIJ_01234'
>>>
>>>
>>> len(x)
17
>>> y = x.lower()
>>> y
'abcde_fghij_01234'
>>>
>>> y = x.upper()
>> y
'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 = 'abcDE_fgHIJ_01234'
>>> x
'abcDE_fgHIJ_01234'
>>>
>>>
>>> len(x)
17
>>> y = x.lower()
>>> y
'abcde_fghij_01234'
>>>
>>> y = x.upper()
>> y
'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
- does not modify x

Python supports a wide variety of string operations

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

Basics of strings

```
>>> x = 'abc'
>>>
>>> x
'abc'
>>>
>>> ",".join(x)
'a,b,c'
>>>
```

str.join(x)

str.join(x): produces a string in which the characters of x have been joined by the string str

does not modify x

Basics of strings

```
>>> x = 'abcdefgh'
>>>
>>> x
'abcdefgh'
>>> 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 = 'abcdefgh'
>>>
>>> x
'abcdefgh'
>>> 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
>>>
```

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

EXERCISE

```
>>> text = "How are you?"
>>>
```

Write code that operates on text and produces the string

'H-O-W- -A-R-E- -Y-O-U-?'

Basics of strings

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

Basics of strings

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

+ applied to strings does concatenation

'*' applied to strings:

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

Basics of strings

```
>>> x = "abcd"
>>> y = 'efgh'
>>> z = 'efgh'
>>>
>>> w = x + y
>>> w
'abcdefgh'
>>>
>>> u = x * 5
>>> u
'abcdabcdabcdabcdabcd'
>>> x - y
Traceback (most recent call last):
  File "<pyshell#39>", line 1, in <module>
    x - y
TypeError: unsupported operand type(s) for -: 'str' and 'str'
>>>
```

+ 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

Basics of strings

```
>>> x = "abcdefg"
>>> y = 'hijk'
>>>
>>> x[3:6]
'def'
>>> x[2:5]
'cde'
>>>
>>> x[:2]
'ab'
>>> x[4:]
'efg'
>>> x[4:] + y[:2]
'efghi'
```

slicing: produces substrings

- characters from 3 (included) to 6 (excluded)
- characters from 2 (included) to 5 (excluded)

- characters from the beginning to 2 (excluded)
- characters from 4 (included) to the end

EXERCISE

```
>>> x = "whoa!"
>>> y = x[2] * len(x)
>>> z = x[3] + x[0] + y
>>> z
```

what is printed here?

EXERCISE

Write an expression that, for any string text, results in the last two characters of text. Assume text has length of 2 or greater.

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

the value read in is represented as a string

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'
>>>
```

the value read in is represented as a string

- string ≡ sequence of characters

- 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 ≡ sequence of characters
- TypeError: indicates an error due to a wrong type

- Fix: explicit type conversion

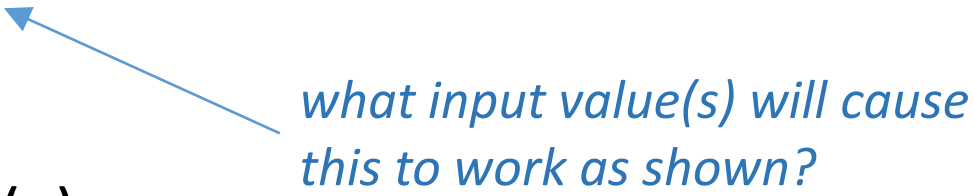
EXERCISE

```
>>> x = input()
12
>>> y = 2*x
```

is this valid?

EXERCISE

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



python review: conditionals

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
>>>
```

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

EXERCISE

Prompt the user for input and assign the result to text.

Set s to the last two characters of text. If text has length less than 2, s should be assigned to an empty string.

Solution

```
text = input()
if len(text) > 2:
    s = text[-2:]
else:
    s = ''
```

python review: while loops

Loops I: **while**

```
>>> n = input('Enter a number: ')
Enter a number: 5
>>> limit = int(n)
>>> i = 0
>>> sum = 0
>>> while i <= limit:
        sum += i
        i += 1

>>> sum
15
>>>
```

Loops I: **while**

```
>>> n = input('Enter a number: ')
Enter a number: 5
>>> limit = int(n)
>>> i = 0
>>> sum = 0
>>> while i <= limit:
        sum += i
        i += 1

>>> sum
15
>>>
```

- **while**-statement syntax:

  while *BooleanExpr* **:**
  $stmt_1$
  …
  $stmt_n$

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

EXERCISE

```
>>> text = "To be or not to be, that is the question."
>>> c = "o"
```

Write the code to count the number of times c occurs in text.

Solution

```
# count the occurrences of c in text
text = "To be or not to be, that is the question.”
c = "o"

n, i = 0, 0
while i < len(text):
    if text[i] == c:
        n += 1
        i += 1
```

python review: lists

Lists

```
>>> 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']
```

Lists

```
>>> 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']
```

a list is a sequence of values

Lists

```
>>> 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']
```

a list 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

Lists

```
>>> 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']
```

a list 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

```
>>> 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']
```

a list 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

```
>>> 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 (+) : similar to strings

multiplication (*) similar to strings

EXERCISE

```
>>> x = [ “abc”, “def”, “ghi”, “jkl” ]
>>> x[1] + x[-1]
```

Lists

```
>>>nums = [18, 3, 24, 63, 18, 4]
>>>num.append(7)
>>>nums
[18, 3, 24, 63, 18, 4, 7]
```

list.append(value)

appends the value to the list.

Lists

```
>>>w = []
>>>w.append(' hello' )
>>>w
[' hello' ]
>>>w.append(' there' ]
>>>w.append(2)
>>>w
[' hello' , ' there' , 2]
```

Empty list

Use append to add additional elements.

Lists

```
>>>w = []
>>>w.append(' hello' )
>>>w
[' hello' ]
>>>w.append(' there' ]
>>>w.append(2)
>>>w
[' hello' , ' there' , 2]
```

Empty list

Use append to add additional elements.

Elements can be of any type

EXERCISE

Write the code to create a list of the even numbers of num. Use a while loop and append.

```
>>> num = [18, 3, 24, 63, 18, 4, 7]
```

Solution

```
# create a list of the even elements of num
nums = [18, 3, 24, 63, 18, 4, 7]

evens = []
while i < len(nums):
    if nums[i] % 2 == 0:
        evens.append(nums[i])
    i += 1
```

Lists: sorting

```
>>> 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]
>>> y
[1, 4, 3, 2, 5]
>>> sorted(y)
[1, 2, 3, 4, 5]
>>> y
[1, 4, 3, 2, 5]
>>>
```

sort() : sorts a list

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

python review: functions

Functions

- **`def`** *fn_name* (arg_1 , …, arg_n):
  - defines a function *fn_name* with n arguments arg_1 , …, arg_n
- **`return`** *expr*
  - optional
  - returns the value of the expression *expr* to the caller
- *fn_name*($expr_1$, …, $expr_n$):
  - calls *fn_name* with arguments $expr_1$, …, $expr_n$

Functions

```
>>> def double(x):
        return x + x
>>> double(7)
14
```

- **`def`** *fn_name* (arg_1 , …, arg_n):
  - defines a function *fn_name* with n arguments arg_1 , …, arg_n

- **`return`** *expr*
  - optional
  - returns the value of the expression *expr* to the caller

Functions

```
>>> def double(x):
        return x + x
>>> double(7)
14
>>>
>>> def num_occurences(text, c):
        n, i = 0, 0
        while i < len(text):
            if text[i] == c:
                n += 1
            i += 1
        return n
>>> num_occurences("To be or not to be, that is the question.", "o")
5
```

- **`def`** *fn_name* (arg_1 , …, arg_n):
  - defines a function *fn_name* with n arguments arg_1 , …, arg_n

- **`return`** *expr*
  - optional
  - returns the value of the expression *expr* to the caller

Lists of Lists

```
>>> x = [ [1,2,3], [4], [5, 6]]
>>> x
[[1, 2, 3], [4], [5, 6]]
>>>
>>>
>>> >>> y = [ ['aa', 'bb', 'cc'], ['dd', 'ee', 'ff'],
['hh', 'ii', 'jj']]
>>> >>> y
[['aa', 'bb', 'cc'], ['dd', 'ee', 'ff'], ['hh', 'ii', 'jj']]
>>>
```

a list can consist of elements of many types, including lists

a list of lists is called a 2-d list

Lists of Lists

```
>>> x = [ [1,2,3], [4], [5, 6]]
>>> x
[[1, 2, 3], [4], [5, 6]]
>>>
>>>
>>> >>> y = [ ['aa', 'bb', 'cc'], ['dd', 'ee', 'ff'], ['hh', 'ii',
'jj']]
>>> >>> y
[['aa', 'bb', 'cc'], ['dd', 'ee', 'ff'], ['hh', 'ii', 'jj']]
>>>
```

a list can consist of elements of many types, including lists

a list of lists is called a 2-d list

if the number of rows and columns are equal, it is a grid

Lists of Lists

```
>>> y
[['aa', 'bb', 'cc'], ['dd', 'ee', 'ff'], ['hh', 'ii', 'jj']]
>>>
>>> y[0]
['aa', 'bb', 'cc']
>>> y[1]
['dd', 'ee', 'ff']
>>> y[2]
['hh', 'ii', 'jj']
>>>
>>> len(y)
3
>>> len(y[0])
3
>>>
```

a list can consist of elements of many types, including lists

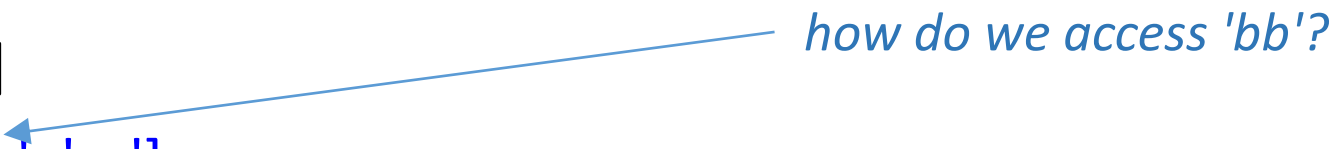
a list of lists is called a 2-d list

if the number of rows and columns are equal, it is a grid

*must check the length of each row

EXERCISE

```
>>> y
[['aa', 'bb', 'cc'], ['dd', 'ee', 'ff'], ['hh', 'ii', 'jj']]
>>>
>>> y[0]
['aa', 'bb', 'cc']
>>> y[1]
['dd', 'ee', 'ff']
>>> y[2]
['hh', 'ii', 'jj']
>>>
```



EXERCISE

```
>>> x = [ [1,2,3], [10,20,30], [100,200, 300]]
>>> x
[[1, 2, 3], [10,20,30], [100,200,300]]
>>>
>>>
```

write the code to sum the first column of x

Helpful hint: first write x out as a grid.
Label the rows

Solution

```
x = [ [1,2,3], [10,20,30], [100,200, 300]]

# sum the first column of a 2-d list x
sum, i = 0, 0
while i < len(x):
        sum = sum + x[i][0]
        i += 1
```

python review: for loops

Loops II: **for**

- The for statement iterates over the items of any sequence (or iterable object) in order
- **for**-statement syntax (the *general form*)

for *Var* **in** *Expr* **:**
    $stmt_1$
    …
    $stmt_n$

- *Expr* is evaluated. $stmt_1$ … $stmt_n$ are executed for each element of the sequence that *Expr* produces; the value each successive element is assigned to *Var* in turn.

Loops II: for

```
>>> nums = [18, 3, 24, 63, 18, 4, 7]
>>> evens = []
>>>
>>> for n in nums:
        if n % 2 == 0:
            evens.append(n)

>>> evens
[18, 24, 18, 4]
>>>
```

sequence: a list or string
(there are more, as you will see)

range

- **range(...)** creates an object that represents a sequence of numbers

- A range can be created in three ways:

  range(*stop*)

  0, 1, ..., stop-1

  range(*start, stop*)

  start, start+1, start+2, ..., stop-1

  range(*start, stop, step*)

  *start, start+step, start+step*2, ..., stop – 1*

- Note that *stop* is always exclusive

for with range

```
>>> nums = [18, 3, 24, 63, 18, 4, 7]
>>> evens = []
>>>
>>> for i in range(len(nums)):
        if nums[i] % 2 == 0:
            evens.append(nums[i])

>>> evens
[18, 24, 18, 4]
>>>
```

represents the sequence 0,1,2,3,4,5,6

EXERCISE

```
>>> grid = [ [18, 25, 36], [23, 25, 18], [20, 54, 7] ]
>>> grid
[ [18, 25, 36], [23, 25, 18], [20, 54, 7] ]
>>>
>>> total = 0
>>> for i in range(len(grid)):
        total += grid[i][0]

>>> total
61
>>>
```

write the code to sum the first column of ***grid*** *using for and range*

EXERCISE

```
>>> grid = [ [18, 25, 36], [23, 25, 18], [20, 54, 7] ]
>>> grid
[ [18, 25, 36], [23, 25, 18], [20, 54, 7] ]
>>>
>>> total = 0
>>>for row in grid:
        total += row[0]

>>>total
61
```

write the code to sum the first column of ***grid*** *using for (no range)*

python review: lists ↔ strings

Strings → lists

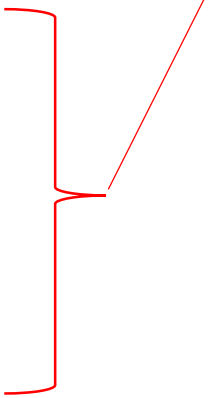
```
>>> names = "John, Paul, Megan, Bill, Mary"
>>> names
'John, Paul, Megan, Bill, Mary'
>>>
>>> names.split()
['John,', 'Paul,', 'Megan,', 'Bill,', 'Mary']
>>>
>>> names.split('n')
['Joh', ', Paul, Mega', ', Bill, Mary']
>>>
>>> names.split(',')
['John', ' Paul', ' Megan', ' Bill', ' Mary']
>>>
```

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

split(*delim*) :
delim, splits the string on *delim*

Lists → strings

```
>>> x = ['one', 'two', 'three', 'four']
>>>
>>> "-".join(x)
'one-two-three-four'
>>>
>>> "!.!".join(x)
'one!.!two!.!three!.!four'
>>>
```



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

returns a string

String trimming

```
>>> x = '    abcd    '
>>>
>>> x.strip()
'abcd'
>>>
>>> y = "Hey!!!"
>>>
>>> y.strip("!")
'Hey'
>>> >>> z = "*%^^stuff stuff stuff^%%%**"
>>>
>>> z.strip("*^%")
'stuff stuff stuff'
```

x.strip() : removes whitespace from both ends of the string *x*

returns a string

x.strip(*string*) : given an optional argument *string*, removes any character in *string* from both ends of *x*

String trimming

Let's look at the built-in documentation for strip:

```
>>> help(str.strip)
strip(...)
    S.strip([chars]) -> str
```

brackets indicate argument is optional
(they are *metacharacters in this doc.*)

Return a copy of the string **S** with leading and trailing whitespace removed.

If **chars** is given and not **None**, remove characters in **chars** instead.

String trimming

Speculate: What do the lstrip() and rstrip() methods do?

```
>>> line = '...testing  \n'
>>> line.rstrip()
'...testing'
>>> line.rstrip().lstrip(".")
'testing'
```

EXERCISE

```
>>> text = "Bear Down, Arizona. Bear Down, Red and Blue."
>>> words = text.split()
>>> words
['Bear', 'Down,', 'Arizona.', 'Bear', 'Down,', 'Red', 'and', 'Blue.']
>>> words_lst = []
>>> for w in words:
        words_lst.append(w.strip("."))

>>> words_lst
['Bear', 'Down', 'Arizona', 'Bear', 'Down', 'Red', 'and', 'Blue']
>>>
```

create a list of words with no punctuation

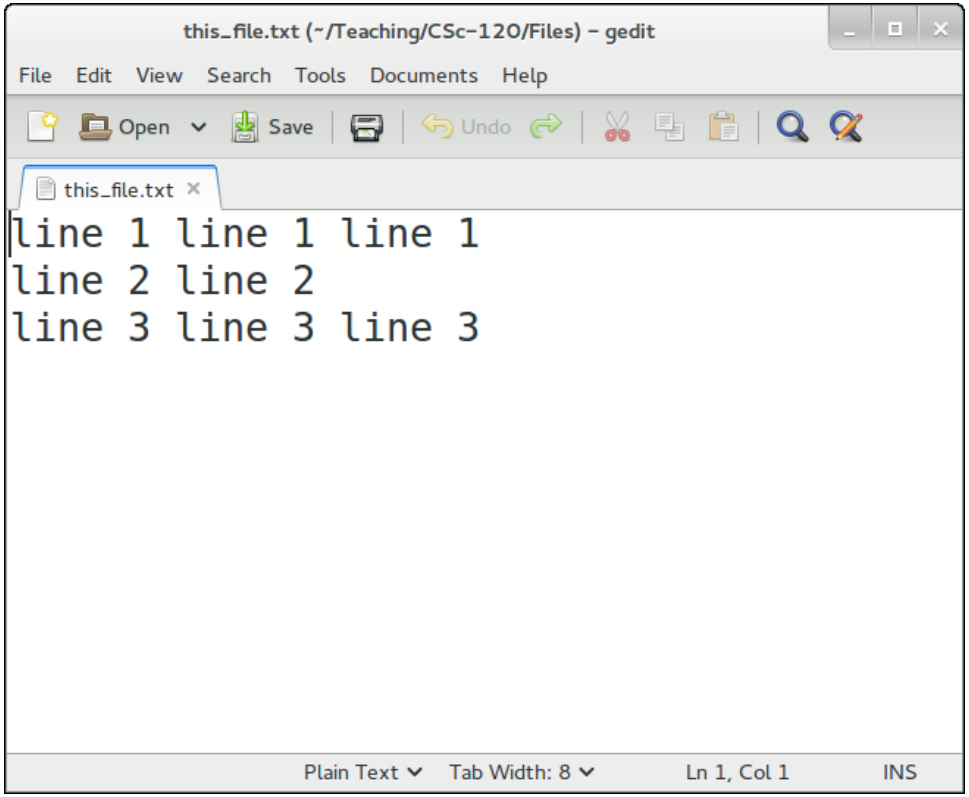
EXERCISE

Do problems 1-3 in the ICA-1 handout.

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

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

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

```
line 1 line 1 line 1

line 2 line 2

line 3 line 3 line 3

>>>
```

Reading user input II: file I/O

```
>>> 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*

Reading user input II: file I/O

```
>>> 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
```

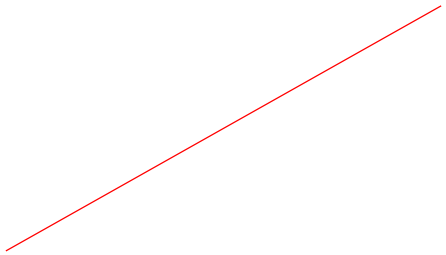
```
>>> print(repr(line))
'line 3 line 3 line 3\n'
```

- *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

```
>>> 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

>>>
```



At this point we've reached the end of the file and there is nothing left to read

Reading user input II: file I/O

```
>>> 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


>>>
>>> infile.close()
>>>
```

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

housekeeping: close the file when we're done with it

Reading user input II: file I/O

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

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

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

>>>
```

Writing output to a file

```
>>> out_file = open("names.txt", "w")
>>>
>>> name = input("Enter a name: ")
Enter a name: Tom
>>>
>>> out_file.write(name  +  '\n')
4
>>> name = input("Enter a name: ")
Enter a name: Megan
>>> out_file.write(name  +  '\n')
6
>>> out_file.close()
>>>
```

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

If the file doesn't exist, is it created.

If it does exist, it is truncated.

Writing output to a file

```
>>> out_file = open("names.txt", "w")
>>>
>>> name = input("Enter a name: ")
Enter a name: Tom
>>>
>>> out_file.write(name  +  '\n')
4
>>> name = input("Enter a name: ")
Enter a name: Megan
>>> out_file.write(name  +  '\n')
6
>>> out_file.close()
>>>
```

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

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

Writing output to a file

```
>>> in_file = open("names.txt", "r")
>>> for line in in_file:
        print(line)
```

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


```
Tom

Megan
```

EXERCISE

Do problem 4 in the ICA-1 handout.

Write a function print_some_words(filename,n) that takes a filename as a string argument and for each line in the file, finds and prints the individual words of *length great than or equal to* n on a separate line.

python review:
a whole program!

Problem

Write a program that prints the number of times one or more specified characters appears in a file.

this_file.txt

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

Interaction:

```
File? this_file.txt
Chars? 123 io
'1': 3
'2': 2
'3': 3
' ': 13
'i': 8
'o': 0
```

Problem decomposition

We'll have three functions:

get_lines(fname)

Read the file named **fname** and return its lines as a list.

count_char(c, lines)

Returns the number of times **c** (a one-character string) appears in **lines**, a list of strings.

main()

Top-level glue

count_chars.py

```
def count_char(c, lines):
    count = 0
    for line in lines:
        for this_char in line:
            if c == this_char:
                count += 1

    return count

def get_lines(fname):
    lines = []
    f = open(fname)
    for line in f:
        lines.append(line)

    f.close()
    return lines
```

count_chars.py, continued

```
def main():
    fname = input("File? ")
    chars = input("Chars? ")

    lines = get_lines(fname)

    for c in chars:
        count = count_char(c, lines)
        print("'" + c + "': " + str(count))

main()
```

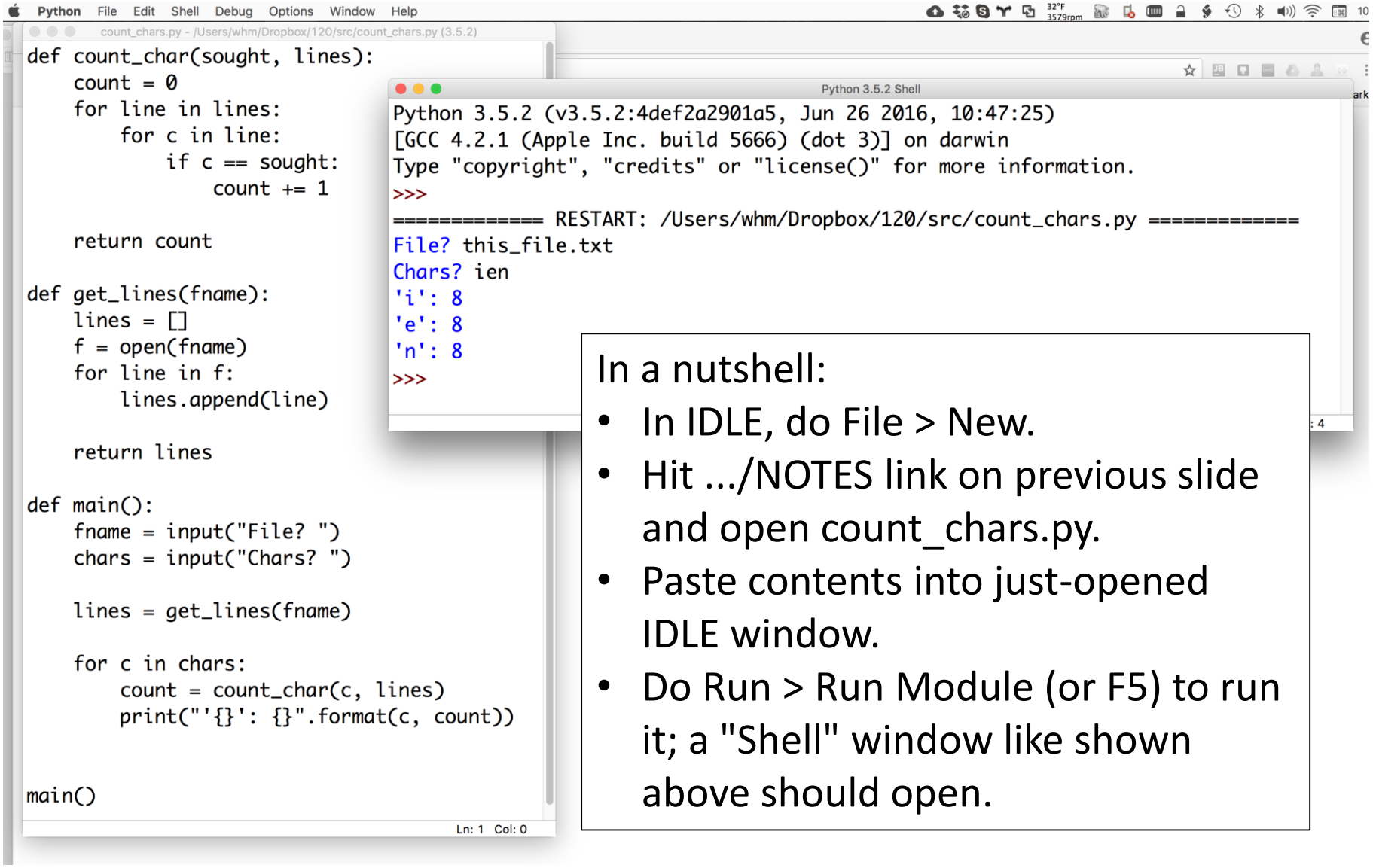
High-level structure of count_chars.py:

```
def count_char(c, lines):
    ...

def get_lines(fname):
    ...

def main():
    ...

main()
```

Notes:

- All code except "main()" is in a function.
- "main()" must be last.
- Function definitions can be in any order.
- What happens if you forget to call main?
- count_chars.py is in http://www2.cs.arizona.edu/classes/cs120/spring19/NOTES/

Running count_chars.py with IDLE



In a nutshell:

- In IDLE, do File > New.
- Hit .../NOTES link on previous slide and open count_chars.py.
- Paste contents into just-opened IDLE window.
- Do Run > Run Module (or F5) to run it; a "Shell" window like shown above should open.

python review: data representation

ASCII codes

- ASCII is "American Standard Code for Information Interchange"
- The ASCII standard specifies numeric codes for 128 characters.
- ASCII was developed in the 1960s
- In 1988 development began on Unicode.
- Version 9 of Unicode can accommodate 1,114,112 "code points".
- The first 128 characters of ASCII and Unicode are the same.
- A simple ASCII chart:

Code	Character
0	NUL (null)
...	...
9	HT (horizontal tab)
10	LF (line feed - new line)
...	...
32	(space)
33	!
34	"
...	...
51	3
52	4
...	...
97	a
98	b
126	~
127	DEL (delete)

ASCII continued

- Python provides ord() and chr() for working with ASCII codes.

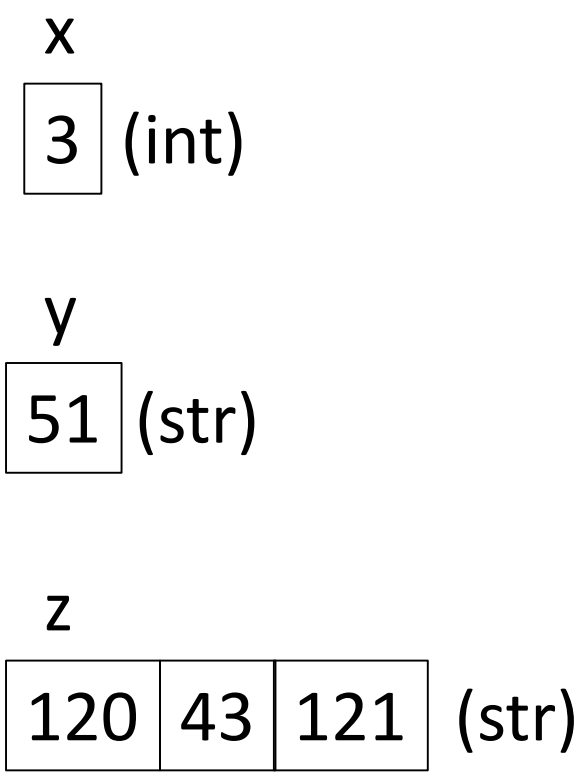
```
>>> ord('a')
97
>>> chr(98)
'b'
>>> print(chr(49),chr(50),chr(51))
1 2 3
>>> ord('\n')
10
```

Code	Character
0	NUL (null)
...	...
9	HT (horizontal tab)
10	LF (line feed - new line)
...	...
32	(space)
33	!
34	"
...	...
51	3
52	4
...	...
97	a
98	b
126	~
127	DEL (delete)

Data representation

- Conceptually, computers store all data as numbers.
- The type of a data value determines the meaning of the number(s) that represent it.

```
>>> x = 3
>>> type(x)
<class 'int'>
>>> y = "3"
>>> type(y)
<class 'str'>
>>> z = "x+y"
```



Data representation

Type is considered when values are compared.

```
>>> a = "5"
>>> b = 5
>>> a == b
False
>>> [120,43,121] == "x+y"
False
>>> chr(120) + chr(43) + chr(121) == "x+y"
True
```

python review: random numbers

EXERCISE

The Python list `student_names` contains the names of all of the students in CSC 120. Write code to select a student to receive a prize.

Note: You don't know about Python's random library.

Random numbers

- How would we solve these problems?
  - model a lottery game
  - model a game of dice
  - select a random element from a list
- Most languages have built-in functions (or a library) for generating random numbers
  - based on an algorithm
  - deterministic, so we say *pseudorandom*
- Python's algorithm
  - Mesenne twister
- See also random.org

The `random` module

- Python's `random` module contains methods for working with random numbers.
- `randint`– generates a random number between two integers, inclusive.

```
>>> random.randint(0,6)
2
```

- Must import `random` first

```
>>> import random
```

Testing trouble!

This program prints three random numbers:

```
import random
def main():
    for i in range(3):
        print(random.randint(1,100))
main()
```

Two runs in IDLE:

```
=== RESTART: rand3.py ===
31
49
26
>>>
=== RESTART: rand3.py ===
64
64
1
>>>
```

What if the program did something complicated, like generating random poetry?

I'd want to be able to get the same sequence of random numbers again and again, so I could get the same poem again and again when testing.

Testing trouble!

We can "seed" Python's random number generator to make it generate the same sequence every time.

```
import random
def main():
    random.seed("7")
    for i in range(3):
        print(random.randint(1,100))

main()
```

Two runs in IDLE:

```
=== RESTART: rand3.py ===
92
73
70
>>>
=== RESTART: rand3.py ===
92
73
70
>>>
```

python review: dictionaries

Dictionaries

- A dictionary is like an list, 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

A Python *dictionary* is like a Python list that can be indexed with values of (almost) any type, not just integers.

Let's make an empty dictionary and experiment with it:

```
>>> d = {}
>>> d
{}
>>> len(d)
0
>>> type(d)
<class 'dict'>
```

Dictionaries

Dictionaries hold pairs of *keys* and *values*.

Let's make a dictionary d add two key/value pairs to it:

```
>>> d = {}
>>> d["seven"] = 7
>>> d["zero"] = 0
>>> d
{'zero': 0, 'seven': 7}
>>> len(d)
2
```

Dictionaries

At hand:

```
>>> d
{'zero': 0, 'seven': 7}
```

Indexing with a key produces its associated value:

```
>>> d["seven"]
7
```

What is produced if a key doesn't exist?

```
>>> d["zeroe"]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: 'zeroe'
```

Dictionaries

The `in` operator can be used to see if a key is in a dictionary:

```
>>> d
{'zero': 0, 'seven': 7}
>>> k = 'zero'
>>> k in d
True
>>> 'x' in d
False
>>> 0 in d
False
```

Dictionaries

It's repetitious to use a series of assignments to populate a dictionary with literal key/value pairs:

```
>>> classrooms= {}
>>> classrooms["CSC 110"] = "ENR2 N120"
>>> classrooms["CSC 120"] = "ILC 120"
>>> classrooms["CSC 372"] = "ILC 119"
```

Alternative:

```
>>> classrooms = { "CSC 110":"ENR2 N120","CSC 120": "ILC 120",
 "CSC 372": "ILC 119"}
>>> len(classrooms)
3
>>> classrooms
{'CSC 110': 'ENR2 N120', 'CSC 120': 'ILC 120', 'CSC 372': 'ILC 119'}
```

EXERCISE

Based on the dictionary material thus far...

1. Think up two simple questions about dictionaries for which you know the answers. (1 min)
2. `for n in range(2):` # (1 min each)

   Ask your neighbor one of your questions.

   Answer one of your neighbor's questions.

EXERCISE

The following code is legal:

```
>>>nums = [2,4,6]
>>> d = {}
>>>d[2] = ‘hello’
>>>d[‘there’] = 14
>>>d[nums] = 3
```

True or False?

keys() and values()

Dictionaries have keys() and values() methods that both produce *iterable objects*.

```
>>> romans = {"I": 1, "V": 5, "X": 10, "L": 50}


>>> romans.keys()
dict_keys(['X', 'I', 'V', 'L'])


>>> romans.values()
dict_values([10, 1, 5, 50])
```

Q: What can we do with an iterable object?

A: Iterate over the values it produces!

EXERCISE

Problem: Write a function print_keys(d) that prints the keys in the dictionary d, one per line.

```
>>> print_keys(classrooms)
CSC 120
CSC 110
CSC 372
>>> print_keys(romans)
X
L
V
I
```

Work with your neighbor(s) and write print_keys(d). (2')

Solution:

```
def print_keys(d):  # NOTES/print_keys.py
    """Print the keys in dictionary d, one per line"""
    for k in d.keys():
        print(k)

# for testing
classrooms = { "CSC 110":"ENR2 N120","CSC 120": "ILC 12
0", "CSC 372": "ILC 119"}

romans = {"I": 1, "V": 5, "X": 10, "L": 50}
```

EXERCISE

Problem: Write print_pairs(d). It prints the key/value pairs in the dictionary d in ascending order of the keys, one per line. Use format in your print statement.

```
>>> print_pairs(classrooms)
CSC 110: ENR2 N120
CSC 120: ILC 120
CSC 372: ILC 119
```

Work with your neighbor(s) and write it! (2')

Hint:

```
>>> help(sorted)
sorted(iterable, key=None, reverse=False)
    Return a new list containing all items from the iterable
    in ascending order.
```

keys() and values()

Solution:

```
def print_pairs(d):  # NOTES/print_pairs.py
    """Print the key/values in dictionary d in
        ascending order of the keys, one per line"""
    for k in sorted(d.keys()):
        print(k, ':', d[k]))

# for testing
classrooms = { "CSC 110":"ENR2 N120", ... }
romans = {"I": 1, "V": 5, "X": 10, "L": 50}
```

keys() and values()

Dictionaries themselves are iterable objects. Observe:

```
>>> romans
{'I': 1, 'V': 5, 'L': 50, 'X': 10}
>>> for x in romans:
        print(x)
I
V
L
X
```

When we iterate over a dictionary what are we doing?

We're iterating over the dictionary's keys.

Problem

Write a function count_chars(s) that returns a dictionary where each key/value pair represents the occurrence count for each unique character found in the string s.

Usage:

```
>>> count_chars("aaa")
{'a': 3}

>>> count_chars("aabaa")
{'a': 4, 'b': 1}

>>> count_chars("to be or not to be")
{'n': 1, 't': 3, 'r': 1, ' ': 5, 'o': 4, 'e': 2, 'b': 2}
```

Pseudocode

Write a function count_chars(s) that takes a string s and returns a dictionary of the counts of all characters in the string.

Pseudocode: (a mix of English and code)

```
def count_chars(s):
    make an empty dictionary counts
        (Each key/value pair represent a character and its count)
    for each character c in s
        if the key c is present
            increment the associated value
        else
            counts[c] = 1
    return counts
```

Prototyping at the shell prompt

A good practice: Work out key computations using the Python shell, especially when you're learning a new feature.

```
>>> counts = {}
>>> s = "abacbacc"
>>> c = s[0]
>>> c in counts
False
>>> counts[c] = 1
>>> counts
{'a': 1}
>>> c = s[1]
>>> c in counts
False
>>> counts[c] = 1
>>> counts
{'a': 1, 'b': 1}
>>> c = s[2]
>>> c in counts
True
>>> counts[c] = counts[c] + 1
>>> counts
{'a': 2, 'b': 1}
```

EXERCISE

Write a function count_chars(s) that takes a string s and returns a dictionary of the counts of all characters in the string.

Starter:

```
def count_chars(s):
    counts = {}

    return counts
```

Work with your neighbor(s) and fill in the middle! (3')

Solution

```
def count_chars(s):  # NOTES/count_chars.py
    """return a dictionary with key/value pairs with
    occurrence counts for the characters in s"""

    counts = {}

    for c in s:
        if not c in counts:    # First occurrence of c
            counts[c] = 1
        else:                  # We've seen c at least once
            counts[c] += 1

    return counts
```

Challenge: See if you can understand NOTES/count_chars2.py

Alternate

- Original

```
for c in s:
    if not c in counts:    # First occurrence of c
        counts[c] = 1
    else:                  # We've seen c at least once
        counts[c] += 1
```

- Alternate

```
for c in s:
    if not c in counts:    # First occurrence of c
        count[c] = 0       # Make an entry
    count[c] += 1
```

Dictionary values can be anything!

Dictionaries can hold values of any type.

```
>>> pairs = {}
>>> pairs["s"] = "a string"
>>> pairs["i"] = 7
>>> pairs["f"] = 3.4
>>> pairs["L"] = [1,2,3]
>>> pairs["n"] = None
>>> pairs["d"] = {"AZ": "Phoenix", "NC": "Raleigh"}


>>> pairs{'f': 3.4, 's': 'a string', 'i': 7, 'n': None, 'd': {'AZ':
'Phoenix', 'NC': 'Raleigh'}, 'L': [1, 2, 3]}
```

Dictionary values can be anything!

At hand:

```
>>> pairs = {}
>>> pairs["d"] = {"AZ": "Phoenix", "NC": "Raleigh"}
```

Let's work with pairs:

```
>>> pairs["d"]
{'AZ': 'Phoenix', 'NC': 'Raleigh'}
>>> pairs["d"]["AZ"]
'Phoenix'
>>> pairs["d"]["NC"]
'Raleigh'
>>> pairs["d"]["NC"][-1]
'h'
```

A dictionary of dictionaries

Let's make some dictionaries:

```
>>> mis_units = { 'mis 101': 4, 'mis 102': 3, 'mis 202': 2 }
>>> csc_units  = { 'csc 110': 4,  'csc 120': 4,  'csc 352': 3 }
>>> ece_units = { 'ece 111': 3, 'ece 222': 3,  'ece 333': 4 }
```

Let's make a dictionary of dictionaries!

```
>>> catalog =
        { "MIS" : mis_units, "CSC" : csc_units, "ECE" : ece_units
}
```

Some people would say that catalog is a "2d-dictionary" .

Others say "two-level dictionary". (First level is departments; second level is courses.)

A dictionary of dictionaries

```
>>> catalog
{'MIS': {'mis 101': 4, 'mis 102': 3, 'mis 202': 2}, 'CSC':
{'csc 110': 4, 'csc 120': 4, 'csc 352': 3}, 'ECE': {'ece
111': 3, 'ece 222': 3, 'ece 333': 4}}


>>> for dept in catalog:
        print(dept, ":", catalog[dept])


MIS : {'mis 101': 4, 'mis 102': 3, 'mis 202': 2}
CSC : {'csc 110': 4,  'csc 120': 4, 'csc 352': 3}
ECE : {'ece 111': 3, 'ece 222': 3, 'ece 333': 4}
>>>
```

Problem

Write a function find_courses(catalog, units) that takes a two-level dictionary 'catalog' and an int 'units' and returns a sorted list of courses having that number of units.

Usage:

```
>>> find_courses(catalog, 4)
['csc 110', 'csc 120', 'ece 333', 'mis 101']
>>> for units in range(2,5):
        print(units, "unit courses:", find_courses(catalog,units))
2 unit courses: ['mis 202']
3 unit courses: ['csc 352', 'ece 111', 'ece 222', 'mis 102']
4 unit courses: ['csc 110', 'csc 120', 'ece 333', 'mis 101']
```

Pseudocode

Spec: find_courses(catalog, units) returns a list of courses in 'catalog' having 'units' units.

A "sketch" of a valid catalog:

```
{'MIS': {'mis 102': 3, ...}, 'CSC': {'csc 110': 4, ...}, 'ECE': {...}}
```

Pseudocode:

```
def find_courses(catalog, units):
    courses = []
    for each department
      for each course in department
        if course's units == units:
          add it to courses
    return sorted courses
```

Work with your neighbor(s) and write find_courses! (3')

Solution

```
def find_courses(catalog, units):   # NOTES/find_courses.py
    courses = []
    for dept in catalog:
        for key in catalog[dept]:
            if catalog[dept][key] == units:
                courses.append(key)
    return sorted(courses)
```

Repetitious!

What questions do you have?

Can it be improved?

Improved

```
def find_courses2(catalog, units): # NOTES/find_courses2.py
    courses = []
    for dept in catalog:
        dept_cat = catalog[dept]
        for course in dept_cat:
            if dept_cat[course] == units:
                courses.append(course)
    return sorted(courses)
```

Introduced an intermediate variable.

- Definitely cleaner
- Maybe faster

What did we change?

Problem

Add a 3 unit course called 'csc 245' to catalog.

```
>>> catalog
{'MIS': {'mis 101': 4, 'mis 102': 3, 'mis 202': 2},
'CSC': {'csc 110': 4, 'csc 120': 4, 'csc 352': 3},
'ECE': {'ece 111': 3, 'ece 222': 3, 'ece 333': 4}}
```

Solution

```
>>> catalog[‘CSC’][csc 245’] = 3
```

Experiment

What's the output?

```
def main():
    d = {}
    for c in "TIP":
        d[c] = c

    for k in d.keys():
        print(k, end=" ")
    print()

main()
```

Output with Python 3.5.2:

```
$ python3.5 dict_order.py
P T I
$ python3.5 dict_order.py
I T P
$ python3.5 dict_order.py
T P I
```

Output with Python 3.6.2:

```
$ python3.6 dict_order.py
T I P
$ python3.6 dict_order.py
T I P
$ python3.6 dict_order.py
T I P
```

IMPORTANT: The insertion order of keys is not guaranteed to be the iteration order! (Even in >= 3.6!)

Dictionary Summary

Operation	Result
{k1:v1, k2: v3, ...}	Dictionary literal. {} is empty dictionary.
len(d)	Return the number of items in the dictionary `d`.
d[key]	Return the item of d with key key. Raises an error if key is not in the dictionary.
d[key] = value	Set `d[key]` to value.
del d[key]	Remove `d[key]` from `d`. Raises an error if key is not in the dictionary. (*not discussed*)
key in d	Return `True` in `d` has a key key, else `False`.
key not in d	Equivalent to not key in `d`.
keys()	Returns an iterable object that will produce all keys
values()	Returns an iterable object that will produce all value
items()	Returns an iterable object that will produce 2-tuples with key/value pairs. (Tuples coming RSN!)

Try `dir({})` and see if you can relate some of the output to the above.

python review: tuples

Tuples ("toople", not "tupple")

A Python tuple is like a Python list that is immutable—a tuple can't be changed.

Let's make a tuple:

```
>>> location = (17.2, 35.9, "Z3")
>>> location
(17.2, 35.9, 'Z3')

>>> type(location)
<class 'tuple'>
```

An item can be fetched with indexing:

```
>>> location[0]
17.2
```

Tuples

An item cannot be assigned to: (tuples are immutable!)

```
>>> location[1] = 23.7
...
TypeError: 'tuple' object does not support item assignment
```

Items cannot be added to or removed from a tuple:

```
>>> location.append(7)
...
AttributeError: 'tuple' object has no attribute 'append'
>>> location.pop(1)
...
AttributeError: 'tuple' object has no attribute 'pop'
```

Tuples

What does the following assignment do?

```
>>> location
(17.2, 35.9, 'Z3')
>>> x, y, sector = location    # parallel assignment
>>> x
17.2
>>> y
35.9
>>> sector
'Z3'
```

```
>>> x = location[0]
>>> y = location[1]
>>> sector = location[2]
```

The assignment above can be called a *destructuring assignment*.
We *unpac*k the tuple.
Style note: When getting multiple values from a tuple, use parallel assignment rather than a series of indexings.

Do we need tuples?

Are tuples just impoverished lists? Do we really need them?

- Using a tuple communicates to the reader that the collection of items is fixed in size and that the items won't change.
  - (0,0) # 2d point
  - (10,-17,-34) # 3d point
  - (5,7,59) # hours, minutes, seconds
  - (10,5,2,5.6) # box dimensions and weight
  - ("Gould-Simpson", 32.229805, -110.9550234)
  - ("upper","left")

Do we need tuples?

Dictionary keys must be immutable values.

- Can we use a tuple as a key?
  - Yes, they are immutable

```
>>> d = {}
>>> d[(0,0)] = "origin"

>>> d
{(0, 0): 'origin'}
```

- Lists cannot be keys because they are mutable.

```
>>> d[[75,98]] = "center"
...
TypeError: unhashable type: 'list'
```

Do we need tuples?

Implementation-wise, tuples have the potential to:

- Require less memory
- Take less time to create than a list
- Allow faster access than lists

Problem

A function can only return one value but sometimes we want that one value to consist of multiple values.

Example:

The function min_max(L) returns the smallest and largest even numbers in L, a list of integers.

What should be the type of the value returned by min_max?

A tuple!

Usage:

```
>>> min_max([5, 10, 3, 4, 7, 12, 18, 1, 25])
(4, 18)
```

EXERCISE

The function min_max(L) returns [a tuple with] the smallest and largest even numbers in L, a list of integers.

[The smallest and largest values are computed using the min() and max() built-in functions.]

Reminder:

```
>>> L = [10,5,7,12,3]
>>> min(L)
3
>>> max(L)
12
```

Work with your neighbor(s) and write min_max. (2 min)

Solution

```
def min_max(L):
    """Returns the smallest and largest even values in L"""
    evens = []
    for num in L:
        if num % 2 == 0:
            evens.append(num)

    return (min(evens),max(evens))
```

Could we return a list instead of a tuple?

Usage:

```
>>> low, high = min_max([5, 10, 3, 4, 7, 12, 18, 1, 25])
>>> print("The range is {}..{}".format(low,high))
The range is 4..18
```

dict.items()

Dictionaries have an items() method that is similiar to the keys() and values() methods.

Speculate: What does dict.items() return?

```
>>> romans
{'V': 5, 'L': 50, 'I': 1, 'X': 10}
>>> romans.items()
dict_items([('V', 5), ('L', 50), ('I', 1), ('X', 10)])
```

print_pairs from revisited, without sorting the keys:

```
def print_pairs2(d):
    for (k,v) in d.items():
        print("{}: {}".format(k, v))
```

Speculate: What would sorted(d.items()) do?

Tuples are sequences

Along with lists, strings, and ranges, tuples are sequences. All of the sequence operations (shown below) can done with tuples.

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

The elements are: *i, i+k, i+2k, ...*

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

Tuples are sequences

Let's try some sequence operations on tuples.

```
>>> t = (10, "twenty", 30.0, [40])

>>> len(t)
4

>>> t2 = t * 2
>>> t2
(10, 'twenty', 30.0, [40], 10, 'twenty', 30.0, [40])

>>> t2[1:-1]
('twenty', 30.0, [40], 10, 'twenty', 30.0)
```

Parentheses often optional

Tuple literals can often be written without parentheses

```
>>> t = 3,4
>>> type(t)
<class 'tuple'>

>>> for item in 3,4,5:
        ...

>>> low,high = min_max([3,4,7,1,8])

def f():
    return 3,4
```

Lists vs. tuples

Thoughts about choosing a list vs. a tuple to store items:

- Needing to store varying numbers of items requires a list.
- Needing to assign to items requires a list.
- Iterating over the all items in a tuple suggests that a list might be a better choice.
- Grouping a fixed number of values, like coordinates in a 3D-point, suggests a tuple.
- A group of a fixed number of dissimilar values, like name, weight, birthday, and address especially suggests a tuple.

But, there are no hard and fast rules. Sometimes the choice is simply a matter of style. Experience helps, too.

EXERCISE

Given

```
>>> x = ( ['aa', 'bb'], ['cc', 'dd'], ['ee'] )
```

Are these assignments legal? If so, how is x changed?

```
>>> x[0] = 'ff'
```

```
>>> x[0][0] = 'ff'
```

```
>>> x[0][0][0] = 'a'
```

Work with your neighbor(s). (1 min)

Solution: mixtures of mutabilities

```
>>> x = ( ['aa', 'bb'], ['cc', 'dd'], ['ee'] )
>>> x[0] = 'ff'
Traceback (most recent call last):
   x[0] = 'ff'
TypeError: 'tuple' object does not support item assignment
```

Tuples are immutable

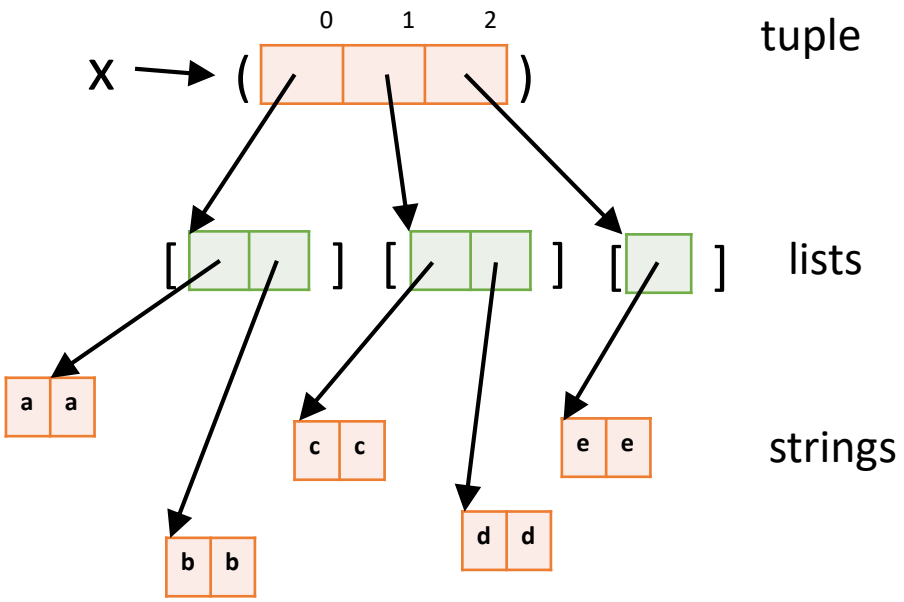
```
>>> x[0][0] = 'ff'
>>> x
(['ff', 'bb'], ['cc', 'dd'], ['ee'])
```

Lists are mutable

```
>>> x[0][0][0] = 'a'
Traceback (most recent call last):
   x[0][0][0] = 'a'
TypeError: 'str' object does not support item assignment
```

Strings are immutable

```
>>> x = ( ['aa', 'bb'], ['cc', 'dd'], ['ee'] )
>>> x[0] = 'ff'
Traceback (most recent call last):
   x[0] = 'ff'
TypeError: 'tuple' object does not
support item assignment
```




```
>>> x[0][0] = 'ff'
>>> x
(['ff', 'bb'], ['cc', 'dd'], ['ee'])
>>> x[0][0][0] = 'a'
Traceback (most recent call last):
   x[0][0][0] = 'a'
TypeError: 'str' object does not
support item assignment
```

Will it work?

Which of the following assignments work?

```
>>>> t = (1,"two",[3,4,5])
>>> t[2][1] = (4,4)
>>> t
(1, 'two', [3, (4, 4), 5])

>>> t2 = 6,7
>>> t[-1].append([t2])
>>> t
(1, 'two', [3, (4, 4), 5, [(6, 7)]])

>>> t2[0] = "six"
...
TypeError: 'tuple' object does not support item assignment
```

Surprise!

Observe:

```
>>> x = [[10,20]]

>>> y = x * 3
>>> y
[[10, 20], [10, 20], [10, 20]]

>>> y[0].append(30)

>>> y
[[10, 20, 30], [10, 20, 30], [10, 20, 30]]
```

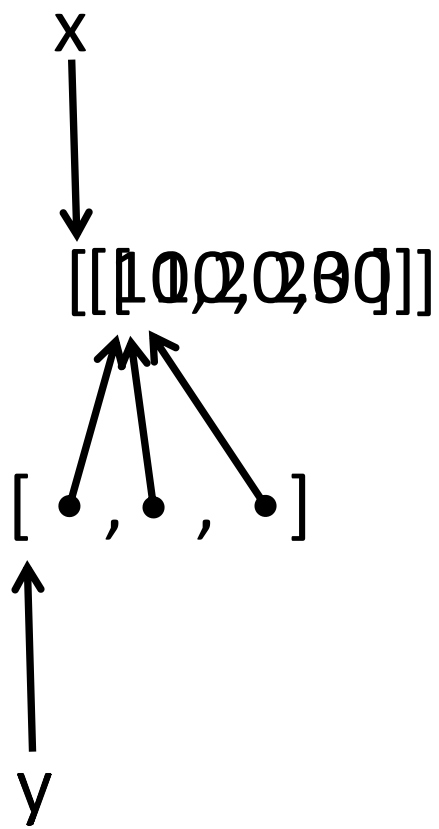
Why??

The list replication (x * 3) created a list with three references to x!

Surprise!

Observe:

```
>>> x = [[10,20]]

>>> y = x * 3
>>> y
[[10, 20], [10, 20], [10, 20]]

>>> y[0].append(30)

>>> y
[[10, 20, 30], [10, 20, 30], [10, 20, 30]]
```



In CPython the id() function returns the address of its argument

```
>>> print(id(x), id(y), id(y[0]), id(y[1]), id(y[2]))
4319621000  4319651848  4319617096  4319617096  4319617096
```

python review: format()

Motivation

Printing a mix of values and literals can be pretty tedious:

```
>>> a, b, c = 10, 'test', 3.4      # parallel assignment

>>> print("a = " + str(a) + ", b = " + b + ", c = " + str(3.4))
a = 10, b = test, c = 3.4
```

Here's another way:

```
>>> print("a = {}, b = {}, c = {}".format(a, b, c))
a = 10, b = test, c = 3.4
```

What is it?

At hand:

```
>>> print("a = {}, b = {}, c = {}".format(a, b, c))
a = 10, b = test, c = 3.4
```

Work with your neighbor(s):

Attempt to explain how the print() statement is being evaluated. In particular:

- What is "format"?
- What type does format produce?
- What are the curly braces doing?

What is it?

At hand:

```
>>> print("a = {}, b = {}, c = {}".format(a, b, c))
a = 10, b = test, c = 3.4
```

- format() is a string method.
- It *interpolates* each argument in turn where {} appears.
- It returns a string. (How would you "prove" that?)

Analogs in other languages:

- printf() in C
- String.format() in Java

count_chars.py improvement

For reference:

```
>>> "{}-{}".format(10,20)
'10-20'
```

Recall this loop from count_chars.py:

```
for c in chars:
    count = count_char(c, lines)
        print("'" + c + "': " + str(count))  # example:      'a': 10
```

Problem: Rewrite the print to use format.

```
>>> print("'{}': {}".format(c, count))
'a': 10
```

format() can do lots!

Here's a sampling of the many kinds of specifications that format() handles:

```
>>> "|{:6d}|>{:^20}<, third = {:7.3f}, {!r}".
    format(100,"center me!",100/3," a ")
"|   100|>    center me!    <, third =  33.333, ' a '"
```

More on format():

https://docs.python.org/dev/library/string.html#format-string-syntax