Python for Informatics

LESSON 4

Files

- File storage allows us to save data that we have processed, so we can access it at a later time.
- File storage is also another source of input data—if we can gain access to a file, we can open, read, and process it.

Opening Files

- Files must be opened before they can be read and processed.

```
fhand = open('C:/UCSD/PythonForInformatics/code/romeo-full.txt')
print(fhand)
<open file 'C:/UCSD/PythonForInformatics/code/romeo-full.txt', mode 'r'
at 0x000000000009B2DD20>
```

- The open function returns a ***file handle***.
- We need the ***file handle*** to perform operations upon the file.

Text Files and Lines

- Text files are a sequence of text lines.
- The character that separates the end of one line and the beginning of the next is the ***newline*** character.
- The newline character is represented in a string literal as an escape sequence, i.e. **\*n***.

```
count_str = 'One!\nTwo!\nFive!\nThree, Sir!'
print(count_str)
One!
Two!
Five!
Three, Sir!
```

How to Read from a File

- Here’s an example of counting the number of lines in a file. Note that each line is discarded as soon as it is read and counted.

```
fhand = open('C:/UCSD/PythonForInformatics/code/romeo-full.txt')
count = 0
for line in fhand :
   count = count + 1
print('Number of lines: ' + str(count))
fhand.close()
Number of lines: 390
```

How to Read from a File

- Assuming you have enough main memory, you can read the entire file all at once.

```
fhand = open('C:/UCSD/PythonForInformatics/code/romeo-full.txt')
inp = fhand.read()
print(str(len(inp)))
print(inp)
fhand.close()
8864
```

How to Read Selectively

- Often it's best to be a picky reader—only read and process lines that meet certain criteria.

```
fhand = open('C:/UCSD/PythonForInformatics/code/romeo-full.txt')
for line in fhand :
    if line.startswith('As') :
        print line
```

```
As daylight doth a lamp; her eyes in heaven

As glorious to this night, being o'er my head

As is a winged messenger of heaven

As that vast shore wash'd with the farthest sea,
```

How to Read Selectively

- The previous example displays to blank lines, the first one being the ***newline*** character at the end of each line, and the second being the newline added by the ***print*** function.
- The rstrip method strips off the whitespace from the right side of the string.

```
fhand = open('C:/UCSD/PythonForInformatics/code/romeo-full.txt')
for line in fhand :
    if line.startswith('As') :
        print line.rstrip()
As daylight doth a lamp; her eyes in heaven
As glorious to this night, being o'er my head
As is a winged messenger of heaven
As that vast shore wash'd with the farthest sea,
```

Skipping What Doesn’t Interest Us

- By structuring our loop to make use of a continue statement, we can skip the lines of input that don’t interest us.

```
fhand = open('C:/UCSD/PythonForInformatics/code/romeo-full.txt')
for line in fhand :
    # Skip lines that don’t interest us.
    if not line.startswith('As') :
        continue
    # Process the lines we want.
    print line.rstrip()
```

```
As daylight doth a lamp; her eyes in heaven
As glorious to this night, being o'er my head
As is a winged messenger of heaven
As that vast shore wash'd with the farthest sea,
```

Skipping What Doesn't Interest Us

- The ***find*** string method can help us find what we are looking for somewhere within our current input line.

```
fhand = open('C:/UCSD/PythonForInformatics/code/romeo-full.txt')
for line in fhand :
    # Skip lines that don't interest us.
    if line.find('love') == -1 :
        continue
    # Process the lines we want.
    print line.rstrip()
```

```
It is my lady, O, it is my love!
O, that I were a glove upon that hand,
Or, if thou wilt not, be but sworn my love,
Call me but love, and I'll be new baptized; <more...>
```

Pick a File, Any File

- Let’s generalize our solution so it can work with any file that the user specifies.
- To simplify, we use the ***%cd*** magic command to position ourselves in the directory where our files are located.

```
%cd C:/UCSD/PythonForInformatics/code
fname = raw_input('Enter the file name: ')
fhand = open(fname)
for line in fhand :
    # Skip lines that don’t interest us.
    if line.find('soft') == -1 :
        continue
    # Process the lines we want.
    print line.rstrip()
Enter the file name: romeo.txt
But soft what light through yonder window breaks
```

Pick a Search String, Any Search String

- Let’s generalize our solution so it can work with any search string that the user specifies.

```
fname = raw_input('Enter the file name: ')
target = raw_input('Enter the search string: ')
fhand = open(fname)
for line in fhand :
    # Skip lines that don’t interest us.
    if line.find(target) == -1 :
        continue
    # Process the lines we want.
    print line.rstrip()
Enter the file name: romeo.txt
Enter the search string: soft
But soft what light through yonder window breaks
```

searcher.py

- Saving this to a Python file, searcher.py, we can run this for any given file and any given search string.

```
python searcher.py
Enter the file name: words.txt

Enter the search string: what
you know how to program, you will figure out what you want
speak to explain to the computer what we would like it to
```

try, *except*, and *open*

- Unfortunately, users can do silly and unexpected things.
- What if the user specifies the name of a file that doesn’t exist?

```
python searcher.py
Enter the file name: spoon.txt


Enter the search string: soup
---------------------------------------------------------------------------
IOError                                   Traceback (most recent call last)
C:\UCSD\PythonForInformatics\code\searcher.py in <module>()
      2 fname = raw_input('Enter the file name: ')
      3 target = raw_input('Enter the search string: ')
----> 4 fhand = open(fname)
      5 for line in fhand :
      6     # Skip lines that don’t interest us.

IOError: [Errno 2] No such file or directory: 'spoon.txt'
```

try, *except*, and *open*

- By adding error handling constructs, we can make our code more robust.

```
fname = raw_input('Enter the file name: ')
target = raw_input('Enter the search string: ')
try:
   fhand = open(fname)
except:
   print('File cannot be opened: ' , fname)
   sys.exit(0)

for line in fhand :
   # Skip lines that don’t interest us.
   if line.find(target) == -1 :
      continue
   # Process the lines we want.
   print line.rstrip()
Enter the file name: spoon.txt
Enter the search string: soup
('File cannot be opened: ', 'spoon.txt')
To exit: use 'exit', 'quit', or Ctrl-D.
An exception has occurred, use %tb to see the full traceback.

SystemExit: 0
```

How to Write to a File

- In order to write to a file, you must open it with mode “w” as the second argument.

```
fname = raw_input('Enter the file name: ')
fhand = open(fname, 'w')
print fhand

line1 = "Remarkable bird, the Norwegian Blue, idn'it, ay?\n"
line2 = "Beautiful plumage!"

fhand.write(line1)
fhand.write(line2)

fhand.close()
Enter the file name: dead_parrot.txt
<open file 'dead_parrot.txt', mode 'w' at 0x0000000009B24E40>

In [34]: more dead_parrot.txt
Remarkable bird, the Norwegian Blue, idn'it, ay?
Beautiful plumage!
```

Lists

- A ***list*** is a ***sequence*** of values.
- The values of a ***list*** are called ***elements*** or ***items***.
- A ***string*** is an example of a ***list***.
- ***A string is a sequence of characters.***
- A new ***list*** can be created using square brackets.

```
comestibles = ['Red Leicester', 'Tilsit', 'Caerphilly', 'Bel Paese']
scores = [88,69, 72, 94]
empty = []
print(comestibles, scores, empty)
(['Red Leicester', 'Tilsit', 'Caerphilly', 'Bel Paese'], [88, 69, 72, 94], [])
```

Lists

- Just as with a ***string***, the elements of a ***list*** are accessed by using brackets and an index value.

```
print(comestibles[2])
Caerphilly

print(scores[0])
88
```

- Unlike ***strings***, ***lists*** are ***mutable***. We can change their values.

```
scores[0] = 87
print(scores)
[87, 69, 72, 94]
```

Lists

- The 0th element of a ***list*** is the first (farthest to the left) element.
- Negative index values wrap around such that the -1th element is the last (farthest to the right) element.

```
print(scores)
[87, 69, 72, 94]
print(scores[0])
87
print(scores[-1])
94
```

Lists

- If you use an index value that is out of range, then you will receive an ***IndexError***.

```
print(scores)
[87, 69, 72, 94]
print(scores[4])
IndexError: list index out of range
```

Language Idioms and the *Pythonic Way*

- Syntactic structures that provide simple, clear, and elegant solutions to common problems are known as language idioms.
- Language idioms in Python are referred to as ***Pythonic***.
- The most common way, the Pythonic way, to traverse a ***list*** is with a ***for*** loop.

```
for comestible in comestibles:
    print(comestible)
Red Leicester
Tilsit
Caerphilly
Bel Paese
```

Lists

- If you attempt to traverse an empty list, there's no element to process, and the body of the loop never executes.

```
for nothing in empty:
    print('Sound in a vacuum')
```

Lists

- The + operator ***concatenates*** or “chains together” one ***list*** to another.

```
p1 = ['Only', 'the', 'finest']
p2 = ['baby', 'frogs']
phrase = p1 + p2
print(phrase)
['Only', 'the', 'finest', 'baby', 'frogs']
```

Lists

- The * operator repeats a ***list*** the given number of times.

```
p3 = p1 * 3
print(p3)
['Only', 'the', 'finest', 'Only', 'the', 'finest', 'Only', 'the', 'finest']
```

Lists

- Previously, we learned how to make ***slices*** of strings.
- Similarly, we can make ***slices of lists***.

```
comestibles = ['Red Leicester', 'Tilsit', 'Caerphilly', 'Bel Paese']
comestibles[0:2]
['Red Leicester', 'Tilsit']
comestibles[2:4]
['Caerphilly', 'Bel Paese']
```

- The ***slice*** begins at the position of the first index, and it ends just before (not including) the position of the second index.
- If the first index is omitted, the ***slice*** begins at the beginning.
- If the second index is omitted, the ***slice*** ends at the end.
- Omitting both indexes creates a full copy of the ***list***.

Lists

- Consider making a copy of a ***list*** before performing an operation that mutates it.

- A ***slice*** on the left side of an assignment can be used to assign to a subset of the ***list***.

```
comestibles = ['Red Leicester', 'Tilsit', 'Caerphilly', 'Bel Paese']
c2 = comestibles
c3 = c2[:]
c2[1:3] = ['Red Windsor', 'Stilton']
print(c2)
['Red Leicester', 'Red Windsor', 'Stilton', 'Bel Paese']
print(comestibles)
['Red Leicester', 'Red Windsor', 'Stilton', 'Bel Paese']
print(c3)
['Red Leicester', 'Tilsit', 'Caerphilly', 'Bel Paese']
```

Lists

- Python ***lists*** also provide a number of useful methods.
  - ***append*** – adds an element to the end of the ***list***
  - ***extend*** – adds the elements of another list to the end of the ***list***.
  - ***sort*** – sorts the ***list*** in ascending order.

Lists

```
print(c3)
['Red Leicester', 'Tilsit', 'Caerphilly', 'Bel Paese']
c3.append('Red Windsor')
print(c3)
['Red Leicester', 'Tilsit', 'Caerphilly', 'Bel Paese', 'Red Windsor']
c4 = ['Danish Blue', 'Brie']
c3.extend(c4)
print(c3)
['Red Leicester', 'Tilsit', 'Caerphilly', 'Bel Paese', 'Red Windsor',
'Danish Blue', 'Brie']
c3.sort()
print(c3)
['Bel Paese', 'Brie', 'Caerphilly', 'Danish Blue', 'Red Leicester',
'Red Windsor', 'Tilsit']
```

Lists

- There are a variety of ways that you can remove elements from a ***list***.
- ***pop*** removes the indexed item from the ***list*** and returns it.

```
print(c2)
['Red Leicester', 'Red Windsor', 'Stilton', 'Bel Paese']
snack = c2.pop(1)
print(snack)
Red Windsor
print(c2)
['Red Leicester', 'Stilton', 'Bel Paese']
```

Lists

- The effect of the ***del*** operator is similar to that of the ***pop*** method, except that it doesn't return anything (actually it returns ***None***).

```
print(c3)
['Bel Paese', 'Brie', 'Caerphilly', 'Danish Blue', 'Red
Leicester', 'Red Windsor', 'Tilsit']
del c3[2]
print(c3)
['Bel Paese', 'Brie', 'Danish Blue', 'Red Leicester', 'Red
Windsor', 'Tilsit']
```

Lists

- The particulary nice thing about the ***del*** operator is that you can supply it with a ***slice***, and it will cut out the specified ***slice*** from the ***list***.

```
print(c3)
['Bel Paese', 'Brie', 'Danish Blue', 'Red Leicester', 'Red
Windsor', 'Tilsit']
del c3[2:5]
print(c3)
['Bel Paese', 'Brie', 'Tilsit']
```

Lists

- What if you don’t know the index value, but you do know the item?
- ***remove*** removes the specified item.

```
print(c2)
['Red Leicester', 'Stilton', 'Bel Paese']
c2.remove('Stilton')
print(c2)
['Red Leicester', 'Bel Paese']
```

- The return value of ***remove*** is ***None***.

Lists

- Python provides an assortment of built-in functions that make it easier to process our ***lists***.
- ***sum()*** returns the sum of the elements of a numeric ***list***.

```
print(scores)
[87, 69, 72, 94]
total = sum(scores)
print(total)
322
```

Lists

- ***max()*** returns the maximum value of the ***list*** elements.
- ***max()*** works with any and all ***lists*** that hold *comparable* elements.
- ***len()*** returns the number of elements in the ***list***.

```
print(scores)
[87, 69, 72, 94]
biggest = max(scores)
count = len(scores)
print(biggest)
print(count)
94
4
```

Lists

- **TRUE** -> A ***string*** is a ***sequence*** of characters.
- **TRUE** -> A ***list*** is a ***sequence*** of values.
- **HOWEVER** -> ***A string is not a list of characters.***
- If you have a ***string***, but you would like a ***list of characters***, you can use the ***list()*** function to make that conversion.

```
castle = 'Camelot'
letters = list(castle)
print(letters)
['C', 'a', 'm', 'e', 'l', 'o', 't']
```

Lists

- If you have a ***string*** that you would like to break down into words or tokens, you can use the ***split()*** function.

```
sentence = "Oh... the cat's eaten it."
words = sentence.split()
print(words)
['Oh...', 'the', "cat's", 'eaten', 'it.']
```

Lists

- By default, ***split()*** breaks your ***string*** down by using whitespace as the ***delimiter***. If you would like ***split()*** to use a different ***delimiter***, you can supply it.

```
sentence = 'This/sounds/like/a/job/for.../Bicycle/Repair/Man'
delimiter = '/'
words = sentence.split(delimiter)
print(words)
['This', 'sounds', 'like', 'a', 'job', 'for...', 'Bicycle', 'Repair', 'Man']
```

Lists

- ***Equivalent*** versus ***identical.***
- **==** tests to see if the referenced objects have the same value—i.e., if they are equivalent.
- ***is*** tests to see if the references refer to the same object—i.e., if they are identical.

```
w1 = 'frog'
w2 = 'frog'
w1 == w2
True
w1 is w2
True
```

Lists

- When you create two ***lists***, they are guaranteed to be independent objects.

```
list1 = [0, 1, 2, 3, 4]
list2 = [0, 1, 2, 3, 4]
list1 == list2
True
list1 is list2
False
```

Lists

- Using two different references to refer to the same underlying object is called ***aliasing***.

```
list1 = [0, 1, 2, 3, 4]
list2 = [0, 1, 2, 3, 4]
list3 = list1
list1 == list3
True
list1 is list3
True
```

- The use of ***aliasing*** with ***mutable objects*** is error prone, and should be avoided.

Lists

- When a ***list*** is passed to a function (or method), it is passed as a reference.

```
def remove_head(lst):
    del lst[0]

list1 = [0, 1, 2, 3, 4]
remove_head(list1)
print(list1)
[1, 2, 3, 4]
```

Lists

- It is important to know if an operator or function is a mutator or not.
- For example, a ***slice*** is not a mutator. ***Slices*** result in the creation of a new ***sequence*** (***list***) object.

```
def remove_head_oops(lst):
    lst = lst[1:]

list1 = [0, 1, 2, 3, 4]
remove_head_oops(list1)
print(list1)
[0, 1, 2, 3, 4]
```

Lists

- Consider a head function that gets the head of the ***list*** but doesn't mutate the ***list***.

```
def head(lst):
   return lst[0:1]

list1 = [0, 1, 2, 3, 4]
item = head(list1)
print(item)
print(list1)
[0]
[0, 1, 2, 3, 4]
```

Lists

- It is best to not create new objects unless it is necessary to do so.

```
def head_with_nlst(lst):
   return lst[0:1]

def head_without_nlst(lst):
   return lst[0]

list1 = [0, 1, 2, 3, 4]
item = head_without_nlst(list1)
item = 42
print(list1)
[0, 1, 2, 3, 4]
```

Software Development Tips

- This week the assigned homework will be more challenging.
- Therefore, here are three tips to help you be more productive and successful.

1. **If in doubt,… do something!**

   Just as writers often encounter the problem of "writer's block," coders encounter the problem of "coder's block." If you are not sure what to do, or you don't even know where to start, *just do something!*

Software Development Tips

2. **Don’t try to solve the whole problem all at once!**

   When confronted with a complex problem, many people make the mistake of thinking that they must understand the entire problem and construct a complete solution before they even get started. Instead, you should break a complex problem into sub-problems, and break those sub-problems into sub-sub-problems. Eventually, you’ll end up with something so simple that you will confidently say, “I know how to code that!”

Software Development Tips

3. **When crafting a solution, ask yourself this question: "What's the simplest thing that could possibly work?"**

   This is a dictum of the *Extreme Programming* software development methodology. More often than not, the simplest solution is the easiest, clearest, least error prone, and most maintainable.