DEPAUL UNIVERSITY
COLLEGE OF COMPUTING AND DIGITAL MEDIA

CSC 243 NOTES
Fall 2020: Amber Settle

Monday, September 21, 2020

Announcements

- Questions about the first quiz?
- The first assignment is due Thursday, September 24th at 9 pm – questions?

Decision structures

The if statement is one of the basic ways that the Python interpreter can select which action to perform, based on a test.

The following is example (**decisions.py**):

```
secret = "Annika"
x = input("Enter x: ")
if x == secret:
      print("You got it!")
print("Done")
```

Exercises

Practice problem 1: Implement a function **reverse**() that inputs a 3-character string from the user and prints the string with its characters reversed. If the user does not provide exactly 3 characters, the function should print the word invalid.

For example:

```
>>> reverse()
Enter a three-character string: and
dna
>>> reverse()
Enter a three-character string: Amber
Invalid
```

Hint: See Table 2.1 on page 25 for string functions that will help with this.

Practice problem 2: Now make the function return a string that consists of the reversed version of the three-character string. If the user does not enter precisely three characters, the function returns the string `'invalid'`

For example:

```
>>> revStr()
Enter a 3-character string: and
```

```
'dna'
>>> s = revStr()
Enter a 3-character string: boo
>>> s
'oob'
>>> s = revStr()
Enter a 3-character string: Amber
>>> s
'invalid'
```

Practice problem 3: Write a function called **login** that asks a user to input a login id, a string, and then checks whether the login id input by the user is in the list [‘emily’, ‘nick’, ‘mark’, ‘julie’]. If so, your function should print “You’re in!”. Otherwise it should print “User unknown”. In both cases, your function should print “Done.” before terminating.

```
>>> login()
Enter login id: julie
You're in!
Done.
>>> login()
Enter login id: amber
User unknown.
Done.
```

Practice problem 4: The Body Mass Index (BMI) is calculated as follows: Multiply a person’s weight (in pounds) by 703 and then divide the result by the square of the person’s height (in inches). A BMI between 19 and 25 is considered optimal.

Write a function **bmi** that takes two input arguments: a person’s weight (in pounds) and height (in inches). Your function should return “Underweight” if the person’s BMI is less than 19, “Optimal” if the person’s BMI is between 19 and 25, and “Overweight” if the person’s BMI is greater than or equal to 25.

See the solutions in **testSolutions.py**.

Digression: Explicitly counting values

Last week we considered the following problem for lists:

Exercise: Consider the following list of student homework grades

```
>>> grades = [9, 7, 7, 10, 3, 9, 6, 6, 2]
```

Write a Python expression to:

- Find the number of 7 grades

We saw that we could use the built-in list function `count()` to do this:

```
>>> grades.count(7)
```

```
2
```

But what if there were no `count()` function? How could we do it?

Write a solution that uses a loop to count.

Formatted output

The **default separator** between items printed using the print function is a blank space.

```
>>> n = 5
>>> r = 5/3
>>> print(n,r)
5 1.66666666667
>>> name = 'Ada'
>>> print(n, r, name)
5 1.66666666667 Ada
```

If we want to insert something else between items printed using the print function, we can use **the optional argument sep** to indicate what the separator should be.

For example, if we wanted to separate the items above with a semicolon instead of a blank space, we would write:

```
>>> print(n, r, name, sep=';')
5;1.66666666667;Ada
```

The print function also supports the formatting argument end.

Normally each successive print function call will print on a separate line:

```
>>> for name in ['Marisa', 'Sam', 'Tom', 'Anne']:
      print(name)

Marisa
Sam
Tom
Anne
```

When the argument end = <some string> is added to the arguments to be printed, the string <some string> is printed after all the arguments to be printed have been.

```
>>> for name in ['Marisa', 'Sam', 'Tom', 'Anne']:
      print(name, end="! ")

Marisa! Sam! Tom! Anne!
```

You can also format output using **formatted strings**.

To do this you construct a string that represents the format of the output.

The variables that should appear in the formatted string are between curly braces.

```
>>> hour = 11
>>> minute = 45
>>> second = 33
>>> f'{hour}:{minute}:{second}'
'11:45:33'
>>> f'{second}:{minute}:{hour}’
'33:11:45'
```

You can also use a formatted string inside of a call to the print function.

```
>>> weekday = "Friday"
>>> month = "March"
>>> day = 10
>>> year = 2017
>>> print(f'{weekday}, {month} {day}, {year} at
{hour}:{minute}:{second}')
Friday, March 10, 2017 at 11:45:33
```

Strings, again

We have a few concepts left to discuss with respect to strings.

String functions

There are functions to convert from integers to strings and back:

```
>>> str(3)
'3'
>>> int('3')
3
>>> str(2.72)
'2.72'
```

The string type supports many **functions to manipulate strings**. Assume that s has been assigned a string value. Then the following are some useful string functions:

- s.**count**(target): Returns the number of occurrences of the substring target in s
- s.**find**(target): Returns the index of the first occurrence of the substring target in s
- s.**lower**(): Returns a copy of the string s converted to lowercase
- s.**upper**(): Returns a copy of the string s converted to uppercase
- s.**isupper**(): Returns True if the string is in upper case
- s.**islower**(): Returns True if the string is in lower case
- s.**replace**(old, new): Returns a copy of the string s in which every occurrence of substring old is replaced by new
- s.**split**(sep): Returns a list of substrings of the string s, obtained using the delimiter string sep; the default delimiter is a space

These functions are demonstrated below:

```
>>> hello = 'hello world'
>>> hello.find("world")
6
>>> hello.replace("world", "universe")
'hello universe'
>>> hello
'hello world'
>>> hello.split()
['hello', 'world']
>>> hello
'hello world'
>>> hello.upper()
'HELLO WORLD'
>>> hello
'hello world'
>>> hello.count('hell')
1
>>> hello.count("o")
2
>>> hello.count("low")
0
```

The method split() is particularly useful for parsing a sentence input as a string into words.

Practice problem: Assign to a variable forecast the string `'`It will be a sunny day today`'` and then write Python statements to do the following:

- Assign to a variable count the number of occurrences of the string 'day' in forecast
- Assign to a variable weather the index where the substring 'sunny' starts
- Assign to a variable change a copy of forecast in which every occurrence of the substring `'`sunny`'` is replaced by `'`cloudy`'`

There are many more **string functions** available.

Single, double, and triple quotes

A **string** is represented as a sequence of characters that is enclosed within either double or single quotes.

To construct a string that contains quotes, we can use the opposite type of quotes, e.g. using single quotes within double quotes or double quotes within single quotes.

```
>>> excuse = 'I am "sick"'
>>> excuse
'I am "sick"'
>>> fact = "I'm sick"
>>> fact
"I'm sick"
```

If text uses both type of quotes, then the escape sequence \’ or \” is used to indicate that a quote is not the string delimiter but is part of the string value.

```
>>> excuse = 'I\'m "sick"'
>>> excuse
'I\'m "sick"'
>>> print(excuse)
I'm "sick"
```

If we want to create **a string that goes across multiple lines**, we can use triple quotes.

```
>>> quote = '''
This poem isn't for you
but for me
after all.
'''
>>> quote
"\nThis poem isn't for you\nbut for me\nafter all.\n"
>>> print (quote)

This poem isn't for you
but for me
after all.
```

File processing (I/O)

Much of computing involves the processing of data stored in files.

A **file** is a sequence of data stored on a secondary memory device such as a disk drive.

A **text file** contains a sequence of characters. You can think of a text file as just a long string, where a newline is indicated by the escape sequence (‘\n’).

Processing a file includes the following:

1. Opening a file for reading or writing
2. Reading from the file and/or writing to the file
3. Closing the file

Why is it necessary to explicitly open and close files from an operating systems point of view?

The short answer is that the operating system needs to manage files so that they remain consistent. One important part of this is making sure that two programs aren’t working on the same file at the same time. Forcing programs to explicitly open and close files allows the operating system to resolve conflicts.

Reading from a file

To **open a file** called example.txt for reading we write:

```
infile = open('example.txt', 'r')
```

The variable `infile` is now associated with an opened file.

There are three methods to read a file that has been associated with the variable name `infile`:

- `infile.read()`: Returns a string containing the remaining, unread contents of the file
- `infile.readline()`: Returns the next line of the file, i.e. the text up to and including the character '\n'
- `infile.readlines()`: Returns a list whose items are the remaining, unread lines of the file.

To close a file associated with the variable `infile`, you write:

```
infile.close()
```

Examples: There are four sample functions in the file **filefunctions.py** that all do the same thing – open a file for reading and then read and output the content of the file.

Let's consider each function as run on the file **example.txt**.

Practice problems:

1. Write a function called **numLines**() that takes one input argument, a file name, and returns the number of lines in the corresponding file.

   It would be used as follows:

   ```
   >>> numLines('example.txt')
   3
   ```

2. Write a function called **numChars**() that takes one input argument, a file name, and returns the number of characters in the file.

   It would be used as follows:

   ```
   >>>numChars('example.txt')
   63
   ```

3. Write a function called **stringCount**() that takes 2 string inputs, a file name and a target string, and returns the number of occurrences of the target string in the file.

It would be used as follows:

```
>>>stringCount('example.txt', 'line')
3
>>> stringCount('example.txt', 'is')
4
>>> stringCount('example.txt', 'the')
1
>>>
```

4. Write a function **getLarger**() that takes a string and a numeric value as parameters and returns the first value in the file that is larger than the numeric value or the numeric value if it is larger than all the values in the file. The file passed as a parameter will only have numeric values in it. It may have more than one numeric value per line.

   It would be used as follows:

```
>>> val = getLarger('nums.txt', 0)
>>> val
1.5
>>> val = getLarger('nums.txt', 10)
>>> val
12
>>> val = getLarger('nums.txt', 13)
>>> val
14
>>> val = getLarger('nums.txt', 14.5)
>>> val
14.5
>>>
```

See **files.py** for the solutions.

Writing to a file

To write to a file, the file must be opened for writing:

```
>>> outfile = open('out.txt', 'w')
```

How does this work?

- If there is no file out.txt in the current working directory, the above open() function will create it.
- If a file out.txt exists, then its contents will be truncated (erased).
- In both cases, the cursor will point to the beginning of the file.

Once a file is opened for writing, the write() function can be used to write strings to it.

The write() function will write the string starting at the cursor position.

```
>>> outfile.write('This is the first line. ')
23
```

The value 23 returned by the write() function is the number of characters written to the file.

The cursor will now point to the position after the period, and the next write will be done starting at that point.

```
>>> outfile.write(' Still the first line …\n')
28
```

Everything written up until the new line character is written to the same line. With the '\n' character written, what follows will go to the second line:

```
>>> outfile.write('Now we are in the second line.\n')
31
```

To write something other than a string, it needs to be converted to a string first:

```
>>> outfile.write('Non string values like ' + str(5)
+ ' must be converted first.\n’)
50
```

Here is where the string format() function is helpful. We print an exact copy of the previous line using it:

```
>>> outfile.write('Non string values like {} must be
converted first.\n'.format(5))
50
```

Finally, if you want to **write to an output file without removing the existing contents**, you need to open the file in a different way:

```
>>> outfile = open('out.txt', 'a')
```

This will open the output file out.txt and append new information to the end of the file. The information already in the file will not be altered.

Printing to a file

The write function is limited in its formatting, which can be frustrating when you’re trying to produce a file that looks attractive.

There is a way to redirect the result of a print to a file.

To do that you open a file for either writing or appending (see above), and then pass print the name of the opened stream to its parameter file.

It will cause the information to be placed into the file. When you're done, close the file.

Example: Suppose you have a list of numbers and you want to write them to a file one per line.

```
>>> lst = [1, 3, 5, 7, 9, 11]
>>> outf = open('out.txt', 'w')
>>> for num in lst:
      print(num, file = outf)


>>> outf.close()
>>>
```

Now the output file `out.txt` will look like:

```
1
3
5
7
9
11
```

Iteration

Sometimes we need to repeat a block of code multiple times, which we achieve by using a loop.

Here we discuss various looping patterns and their usage. Then we will see the while loop.

Loop patterns: Iteration loops

Iterating through an explicit, ordered sequence of values and performing some action for each value is the simplest pattern for a for loop.

Reading a file in Python also uses the iteration loop pattern.

The following code opens the file 'test.txt' for reading, and then reads and prints the contents of the file. Recall that we can read a file by assigning its text contents into a string or into a list of lines of the file.

In this case we use the first approach and iterate through the string characters and print each one.

```
def f1():
    infile = open('test.txt', 'r')
    s = infile.read()
    for character in s:
        print(character, end="")
    infile.close()
```

In the next case, we iterate through a list of strings:

```
def f2():
    infile = open('test.txt', 'r')
    l = infile.readlines()
    for line in l:
        print(line, end="")
    infile.close()
```

Python provides a shortcut to the above technique:

```
def f3():
    infile = open('test.txt', 'r')
    for line in infile:
        print(line, end="")
    infile.close()
```

See all of these implementations in **loops.py**.

Loop patterns: Indexed loops

One special but important case of the iteration loop pattern is the **iteration over a list of consecutive integers**.

A very common reason to iterate over a sequence of consecutive integers is to **generate indices of values in a list or characters in a string**.

Recall the first iteration loop example:

```
>>> l = ['cat', 'dog', 'chicken']
>>> for animal in l:
      print(animal)

cat
dog
chicken
```

Instead of iterating through the values of the list l, we could have iterated through the indices of l and achieved the same thing:

```
>>> for i in range(len(l)):
      print(l[i])

cat
dog
chicken
```

Note how the range and len functions are used to generate the list [0, 1, 2], that is, the list of valid indices of l.

Iterating through indices is more complex and less intuitive than using the for loop to move through elements of a list.

Why would we use it?

Sometimes it makes more sense to iterate through a sequence by index rather than by value.

For example, consider the problem of checking whether a list l is sorted in non-decreasing order.

To do this, it suffices to check whether each value in the list is greater than or equal to the previous one, if there is a previous one.

Doing this without using indices seems tricky:

```
>> for i in l:
      # now compare i with the previous value
```

The problem here is that we don't have a way to easily get the previous value in the list.

If we iterate through the list by index, it becomes much clearer how to do it:

```
>>> for i in range(len(l)):
      if l[i-1] > l[i]:
            print('l is not in sorted order')
```

Unfortunately, there is a mistake in the code above. If we are looking at the first element, that is i = 0, we will compare l[0-1] = l[-1] with l[0]. But l[-1] doesn't exist.

```
>>> for i in range(1, len(l)):
      if l[i-1] > l[i]:
            print('l is not in sorted order')
```

See the solution in the function **sorted**() in the file **loopSols.py**.

Practice problem: Write a function **dupChars**() that takes a string as parmeter and returns True if there are any adjacent duplicated characters and False otherwise.

It would be used as follows:

```
>>> dupChars('Jaakko')
True
>>> dupChars('Arto')
False
>>> dupChars('test')
False
>>>
```

See the solution in the file **loopSols.py**.