Meeting 6 Exercises

Info 206

12 September 2017

Write individual scripts to solve the following exercises.

1. File Input and Output

In many data applications, your first step will be to read data from a file. Likewise, you will often need to write data or results to an output file. In this section, we will show you how to read and write files using Python.

Let's first look at the fundamental command that will enable you to access a file: `open`.

```
fileHandle = open(path_to_file, mode)
```

The `open` function returns a Python object of type file; in this case we assign that file to the variable `fileHandle`. This is the variable we will use to read or manipulate the contents of the file.

The `path_to_file` variable specifies where the file is located. This is a fully qualified path and file name that tells Python where to look for the file on your computer.

The mode parameter specifies what kind of operations that we want to do on the file, and what kind of file it is.

The first letter indicates the operation: - r means read - w means write, and overwrite the file if it exists - x means write, but only if the file does not already exist - a means write (append): write at the end of the file if it exists

The second letter of mode indicates the file type: - t (or nothing) means text - b means binary

After opening a file and manipulating it in some way, you should remember to close the file as well. Closing the file ensures that Python does not maintain file handles to files it is not using anymore. This practice can keep memory usage low as well.

Writing a Text File Using `write()`

Let's write a file using Python. First we'll need some content.

```
file_content = '''The time will come
when, with elation,
you will greet yourself arriving
at your own door, in your own mirror,
and each will smile at the other's welcome,
and say, sit here. Eat.
You will love again the stranger who was your self.
Give wine. Give bread. Give back your heart
to itself, to the stranger who has loved you

all your life, whom you ignored
for another, who knows you by heart.
Take down the love letters from the bookshelf,

the photographs, the desperate notes,
peel your own image from the mirror.
Sit. Feast on your life.'''
```

We want to create a new file using the contents in our multiline string. To do that, we will follow three steps: open the file, write to it, then close it.

```
poem_file = open('poem.txt', 'wt')
poem_file.write(file_content)
poem_file.close()
```

We programatically created the file poem.txt with the given content. If the file already existed, we overwrote it with our new content.

Next, let's append more text to the file. We will need to open it using the "a" mode to append.

```
poem_file = open('poem.txt', 'at')
poem_file.write("\n--Written By Derek Walcott")
poem_file.close()
```

Notice the newline character \n in the string we passed to the write method. The function `write()` enters all characters verbatim, so we have to explicitly include a newline if we want to move to the next line.

Read a Text File With `read()`, `readline()`, or `readlines()`

Reading files is easy in Python, but there are a few important things to keep in mind.

First, the `read()` function with no arguments will load the entire file into memory. This is acceptable for small files, but files can get quite large. If a file is large enough, it can cause your application to run out of memory. Still, if you know you have a small file, this is an easy way to read its contents.

```
poem_file_read = open('poem.txt', 'rt')
poem_read = poem_file_read.read()
poem_file_read.close()
print(poem_read)
```

You can also use the function `readlines()` that will also load all of the file in memory but conveniently return a list in which each item is a line in the file.

```
poem_file_read = open('poem.txt', 'rt')
poem_read_array = poem_file_read.readlines()
poem_file_read.close()
print(poem_read_array)

print(poem_read_array[0])
```

Notice the contents of the file are now in a list. You can now process a file line by line like this:

```
for line in poem_read_array :
    print(line)
```

We have a `for` loop that lets us access each line of the file one by one and process it. This is a very common procedure when working with text. Even though our file is divided into separate lines, our code still loads the entire file into memory when we call `readlines()`.

The `readline()` function saves us from doing this. This method reads a single line from the file and then stops. If we call it again later, it will continue by returning the next line in the file.

```
poem_file_read = open('poem.txt', 'rt')

while True:
    line = poem_file_read.readline()
    if not line:
        break

    print(line)

poem_file_read.close()
```

Notice that we use an infinite loop to read lines one at a time using the ‘readline()’ function. When we reach the end of the file, `readline()` will return an empty string. When this happens, we break out of the loop. The advantage of this technique is that we read our file one line at a time and do not have to worry about memory issues (unless we encounter an unusually long line).

There is an even easier way to read files. The file handle itself is an iterator so you can pass it directly to a `for` statement.

```
poem_file_read = open('poem.txt', 'rt')

for line in poem_file_read:
    print(line)

poem_file_read.close()
```

Even more Pythonic code would be the following:

```
with open('poem.txt', 'rt') as f:
    for line in f:
        print(line)
```

a. Create a script, text_[lastname].py that creates a file, poem_[lastname].txt, that contains your favorite poem (ideally a short one!). Follow the steps above to write the text, add a line that includes the author, and use the while..for method (final method above) to open the file and print each line of your file.

b. Add code to the file that prints each line of the poem using list comprehension.

---

2. Exhaustive Search

Exhaustive search is an example of a *brute force algorithm*. It searches through all possible solutions until the right one is found. You can see that the brute force algorithm will always find the answer, but it can take a rather long time. Depite this, there are many times in programming when you will want to use a brute force algorithm. It could be that you know the number of possible solutions is small, so the brute force algorithm will work in a reasonable time. Brute force algorithms are also very simple. Finally, it could just be that the problem space is unstructured and there is no better choice available.

Exhuastive Search to Find a Square Root

```
x = float(input("enter a number: "))
epsilon = 0.00001
num_guesses = 0
ans = 0.0

while ans * ans <= x:
    [ADD CODE HERE] # add a step to our guess (ans)
    [ADD CODE HERE] # add to the counter for number of guesses (num_guesses)

```

```
print('number of guesses =', num_guesses)
print(ans, 'is close to square root of', x)
```

a. Complete the algorithm above to perform exhaustive search. Save the file as a script exhaustivesearch_[lastname].py

b. Try finding the square root of 10 with this algorithm. What do you find?

c. Now try the program on a large number, say 12345. How long does it take?

d. What happens when you change the value for eplison to a larger number?

---

3. Bisection Search

Here is another algorithm that is well known among computer scientists: a *bisection search*. Instead of making a single guess, imagine a window, or an interval of numbers that we know contains the square root of x. At the beginning, we can use the window from 0 to x. Clearly, the square root of x has to be in there somewhere.

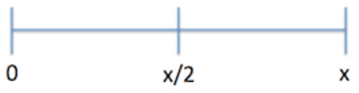

Figure 1: bisection1

Next, let's look at the midpoint of our window, which will be $x/2$ at the moment. We can check if this point is too low or too high by squaring it and comparing it to x. If the midpoint of our window is too low, we can throw away the bottom half of the window and focus on the top half. Otherwise, we throw out the top half of the window and focus on the bottom half.

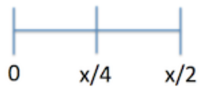

Figure 2: bisection2

Suppose that our first midpoint was too high. Now we have a new window, which ranges from `0` to `x/2`, and we can repeat the same process. Eventually our window will be small enough that we know the midpoint is within epsilon of the right answer.

he bisection search is appropriate for a wide range of computing problems. It leverages the fact that our solution space is ordered. We know that if our guess squared is bigger than x, then anything bigger than our guess is also too big. By leveraging this structure, the bisection search can complete much faster than a brute force algorithm. When you see a solution space that's ordered in this way, consider where a bisection search could be adapted to solve your problem.

The Python script below implements a bisection search. Notice that we use variables low and high to keep track of the bottom and the top of our window. When the size of the window shrinks below 2 * epsilon, we know that any point inside the window is within epsilon of the midpoint. This means that the midpoint is a valid guess for the square root of x, up to the precision we want.

Bisection Search to Find a Square Root

```
x = float(input("enter a number:"))
epsilon = 0.00001
```

```python
num_guesses = 0
low = 0.0
high = x
ans = (high + low)/2.0

while high-low >= 2 * epsilon:
    print("low =", low, "high =", high)
    [ADD CODE HERE]

print('number of guesses =', num_guesses)
print(ans, 'is close to square root of', x)
```

a. Complete the algorithm above to perform exhaustive search. Save the file as a script exhaustivesearch_[lastname].py

b. Try running the algorithm to find the square root of 10. How many steps did the algorithm take? What is the final guess?

c. Try running the algorithm to find the square root of 26894. How many steps did the algorithm take? What is the final guess?

d. Now try running the algorithm to find the square root of 0.25. Does anything go wrong? How could you fix the algorithm so it works correctly?