Week 6

`for` loops over indices

range

Here is the top part of the help for `range`:

```
class range(object)
  | range([start,] stop[, step]) -> range object
  |.
  | Returns a virtual sequence of numbers from start to stop by step.
```

The `stop` value is not included.

`range` is typically used in a `for` loop to iterate over a sequence of numbers. Here are some examples:

```
# Iterate over the numbers 0, 1, 2, 3, and 4.
for i in range(5):

# Iterate over the numbers 2, 3, and 4.
for i in range(2, 5):

# Iterate over the numbers 3, 6, 9, 12, 15, and 18.
for i in range(3, 20, 3):
```

Iterating over the indices of a list

Because `len` returns the number of items in a list, it can be used with `range` to iterate over all the indices. This loop prints all the values in a list:

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

This also gives us flexibility to process only part of a list. For example, We can print only the first half of the list:

```
for i in range(len(lst) // 2):
    print(lst[i])
```

Or every other element:

```
for i in range(0, len(lst), 2):
    print(lst[i])
```

Not "What" but "Where"

Previously, we have written loops over characters in a string or items in a list. However, sometimes there are problems where knowing the value of the items in a list or the characters in a string is not enough; we need to know where it occurs (i.e. its index).

Example 1

The first example is described below:

```
def count_adjacent_repeats(s):
    ''' (str) -> int

    Return the number of occurrences of a character and an adjacent character
    being the same.

    >>> count_adjacent_repeats('abccdeffggh')
    3
```

```
    '''

    repeats = 0

    for i in range(len(s) - 1):
        if s[i] == s[i + 1]:
            repeats = repeats + 1

    return repeats
```

We want to compare a character in the string with another character in the string beside it. This is why we iterate over the indices because the location is important, and only knowing the value of the character does not provide us with enough information. This is how we are able to count repeated characters in a string. We can't execute the body of the loop if `i` is `len(s) - 1` because we compare to `s[i + 1]`, and that would produce an `IndexError`.

Example 2

The second example is described below:

```
def shift_left(L):
    ''' (list) -> NoneType

    Shift each item in L one position to the left and shift the first item to
    the last position.

    Precondition: len(L) >= 1

    >>> shift_left(['a', 'b', 'c', 'd'])
    '''

    first_item = L[0]

    for i in range(len(L) - 1):
        L[i] = L[i + 1]

    L[-1] = first_item
```

For the same reasons as above, merely knowing the value of the items in the list is not enough since we need to know where the items are located; we need to know the index (position) of the item in the list.

LECTURE CODE

```
def shift_left(L):
    ''' (list) -> NoneType

    Shift each item in L one position to the left
    and shift the first item to the last position.

    Precondition: len(L) >= 1
    '''

    first_item = L[0]

    for i in range(1, len(L)):
        L[i - 1] = L[i]

    L[-1] = first_item


def count_adjacent_repeats(s):
    ''' (str) -> int

    Return the number of occurrences of a character and
    an adjacent character being the same.
```

```
    >>> count_adjacent_repeats('abccdeffggh')
    3
    '''

    repeats = 0

    for i in range(len(s) - 1):
        if s[i] == s[i + 1]:
            repeats = repeats + 1

    return repeats
```

Video 2: Parallel Lists and Strings

Correspondings Elements

Two lists are *parallel* if they have the same length and the items at each index are somehow related. The items at the same index are said to be at *corresponding* positions.

Consider these two lists:

```
list1 = [1, 2, 3]
list2 = [2, 4, 2]
```

In these two lists, the corresponding element of `list1[0]` is `list2[0]`, the corresponding element of `list2[1]` is `list1[1]`, and so on.

Example of Corresponding Elements

```
def match_characters(s1, s2):
    ''' (str, str) -> int

    Return the number of characters in s1 that are the same as the character
    at the corresponding position of s2.

    Precondition: len(s1) == len(s2)

    >>> match_characters('ate', 'ape')
    2
    >>> match_characters('head', 'hard')
    2
    '''

    num_matches = 0

    for i in range(len(s1)):
        if s1[i] == s2[i]:
            num_matches = num_matches + 1

    return num_matches
```

The function above counts the corresponding elements of the two strings that are the same character. If a character of `s1` at index i is the same as the character of `s2` at the same index, then we increment `num_matches` by 1 (since they match). Otherwise, we continue on to the next pair of corresponding elements and compare them.

LECTURE CODE

```
def count_matches(s1, s2):
    ''' (str, str) -> int

    Return the number of positions in s1 that contain the
    same character at the corresponding position of s2.
```

```
    Precondition: len(s1) == len(s2)

    >>> count_matches('ate', 'ape')
    2
    >>> count_matches('head', 'hard')
    2
    '''

    num_matches = 0

    for i in range(len(s1)):
        if s1[i] == s2[i]:
            num_matches = num_matches + 1

    return num_matches


def sum_items(list1, list2):
    ''' (list of number, list of number) -> list of number

    Return a new list in which each item is the sum of the items at the
    corresponding position of list1 and list2.

    Precondition: len(list1) == len(list2)

    >> sum_items([1, 2, 3], [2, 4, 2])
    [3, 6, 5]
    '''

    sum_list = []

    for i in range(len(list1)):
        sum_list.append(list1[i] + list2[i])

    return sum_list
```

Video 3: Nested Lists

Lists can contain items of any type, including other lists. These are called *nested lists*.

Here is an example.

```
>>> grades = [['Assignment 1', 80], ['Assignment 2', 90], ['Assignment 3', 70]]
>>> grades[0]
['Assignment 1', 80]
>>> grades[1]
['Assignment 2', 90]
>>> grades[2]
['Assignment 3', 70]
```

To access a nested item, first select the sublist, and then treat the result as a regular list.

For example, to access `'Assignment 1'`, we can first get the sublist and then use it as we would a regular list:

```
>>> sublist = grades[0]
>>> sublist
['Assignment 1', 80]
>>> sublist[0]
'Assignment 1'
>>> sublist[1]
80
```

Both `sublist` and `grades[0]` contain the memory address of the `['Assignment 1', 80]` nested list.

We can access the items inside the nested lists like this:

```
>>> grades[0][0]
'Assignment 1'
>>> grades[0][1]
80
>>> grades[1][0]
'Assignment 2'
>>> grades[1][1]
90
>>> grades[2][0]
'Assignment 3'
>>> grades[2][1]
70
```

Nested Loops

Bodies of Loops

The bodies of loops can contain any statements, including other loops. When this occurs, this is known as a *nested loop*.

Here is a nested loop involving 2 `for` loops:

```
for i in range(10, 13):
    for j in range(1, 5):
        print(i, j)
```

Here is the output:

```
10 1
10 2
10 3
10 4
11 1
11 2
11 3
11 4
12 1
12 2
12 3
12 4
```

Notice that when `i` is 10, the inner loop executes in its entirety, and only after j has ranged from 1 through 4 is `i` assigned the value `11`.

Example of Nested Loops

```
def calculate_averages(grades):
    ''' (list of list of number) -> list of float

    Return a new list in which each item is the average of the grades in the
    inner list at the corresponding position of grades.

    >>> calculate_averages([[70, 75, 80], [70, 80, 90, 100], [80, 100]])
    [75.0, 85.0, 90.0]
    '''

    averages = []

    # Calculate the average of each sublist and append it to averages.
    for grades_list in grades:

        # Calculate the average of grades_list.
        total = 0
        for mark in grades_list:
            total = total + mark
```

```
        averages.append(total / len(grades_list))

    return averages
```

In `calculate_averages`, the *outer* `for` loop iterates through each sublist in `grades`. We then calculate the average of that sublist using a `nested`, or `inner`, loop, and add the average to the accumulator (the new list, `averages`).

LECTURE CODE

```
def averages(grades):
    '''
    (list of list of number) -> list of float

    Return a new list in which each item is the average of the
    grades in the inner list at the corresponding position of
    grades.

    >>> averages([[70, 75, 80], [70, 80, 90, 100], [80, 100]])
    [75.0, 85.0, 90.0]
    '''

    averages = []

    for grades_list in grades:
        # Calculate the average of grades_list and append it
        # to averages.

        total = 0
        for mark in grades_list:
            total = total + mark

        averages.append(total / len(grades_list))

    return averages
```

Video 5: Reading Files

Information stored in files can be accessed by a Python program. To get access to the contents of a file, you need to open the file in your program. When you are done using a file, you should close it.

Opening and Closing A File

Python has a built-in function open that can open a file for reading.

The form of open is `open(filename, mode)`, where mode is `'r'` (to open for reading), `'w'` (to open for writing), or 'a' (to open for appending to what is already in the file).

This opens a file called `In Flanders Fields.txt` for reading:

```
flanders_file = open('In Flanders Fields.txt', 'r')
```

Note that if the file is saved in the same directory as your program, you can simply write the name of the file, as what was done in the above example. However, if it is not saved in the same directory, you must provide the path to it.

To close a file, you write `flanders_file.close() .`

> In Flanders Fields
>
> In Flanders fields the poppies blow
> Between the crosses, row on row,
> That mark our place; and in the sky

> The larks, still bravely singing, fly
> Scarce heard amid the guns below.
>
> We are the Dead. Short days ago
> We lived, felt dawn, saw sunset glow,
> Loved and were loved, and now we lie
> In Flanders fields.
>
> Take up our quarrel with the foe:
> To you from failing hands we throw
> The torch; be yours to hold it high.
> If ye break faith with us who die
> We shall not sleep, though poppies grow
> In Flanders fields.
>
> -John McCrae

There are four standard ways to read from a file. Some use these methods:

`readline():` read and return the next line from the file, including the newline character (if it exists). Return the empty string if there are no more lines in the file.

`readlines():` read and return all lines in a file in a list. The lines include the newline character.

`read():` read the whole file as a single string.

Approach	Code	When to use it

The `readline` approach	file = open(filename, 'r')
	# Read lines until we reach the
	# place in the file that we want.
	line = file.readline()
	while we are not at the place we want:
	  line = file.readline()
	
	# Now we have reached the section
	# of the file we want to process.
	
	line = file.readline()
	while we are not at the end of the section:
	  process the line
	  line = file.readline()
	
	flanders_file.close()
		When you want to process
		only part of a file.

The `for` *line* in *file* approach	file = open(filename, 'r') for line in file: process the line file.close()	When you want to process every line in the file one at a time.
The `read` approach	file = open(filename, 'r') contents = file.read() now process contents file.close()	When you want to read the whole file at once and use it as a single string.

The `readlines` approach	file = open(filename, 'r')
	
	# Get the contents as a list of strings.
	contents_list = file.readlines()
	
	process contents_list using indexing to access particular lines from the file
	
	file.close()
		When you want to examine each line of a file by index.

Examples from the video

Here are the code examples that appeared in the video. All of them read the entire file into a string and print that string.

The `readline` approach

```
flanders_file = open(flanders_filename, 'r')
flanders_poem = ''

line = flanders_file.readline()
while line != "":
    flanders_poem = flanders_poem + line
    line = flanders_file.readline()

print(flanders_poem)
flanders_file.close()
```

The `for` line in file approach

```
flanders_file = open(flanders_filename, 'r')
flanders_poem = ''

for line in flanders_file:
    flanders_poem = flanders_poem + line

print(flanders_poem)
flanders_file.close()
```

The `read` approach

```
flanders_file = open(flanders_filename, 'r')
flanders_poem = flanders_file.read()

print(flanders_poem)
flanders_file.close()
```

The `readlines` approach

```
flanders_file = open(flanders_filename, 'r')
flanders_poem = ''

flanders_list = flanders_file.readlines()
for line in flanders_list:
    flanders_poem = flanders_poem + line

print(flanders_poem)
flanders_file.close()
```

Video 6: Write Files

Writing To A File Within A Python Program

In order to write to a file, we use `file.write(str)`. This method writes a string to a file. Method `write` works like Python's print function, except that it does not add a newline character.

File dialogs

Module `tkinter` has a submodule called `filedialog`. We import it like this:

```
import tkinter.filedialog
```

Function askopenfilename asks the user to select a file to open:

```
tkinter.filedialog.askopenfilename()
```

This function returns the full path to the file, so we can use that when we call function open to open that file.

```
from_filename = tkinter.filedialog.askopenfilename()
```

Function asksaveasfilename asks the user to select a file to save to, and provides a warning if the file already exists.

```
to_filename = tkinter.filedialog.asksaveasfilename()
```

Example

Below is a program that copies a file, but puts "Copy" as the first line of the copied file.

In order to prompt a user for a file.

Now we can open the file we want to read from and get the contents:

```
from_file = open(from_filename, 'r')
contents = from_file.read()
from_file.close()
```

And we can open the file we want to write to and write the contents:

```
to_file = open(to_filename, 'w')
to_file.write('Copy\n')  # We have to add the newline ourselves.
to_file.write(contents)  # Now write the contents of the file.
to_file.close()
```

Video 7

Grade

```
def read_grades(gradefile):
    '''(file open for reading) -> list of float

    Read and return the list of grades in gradefile.

    Precondition: gradefile starts with a header that contains
    no blank lines, then has a blank line, and then lines
    containing a student number and a grade.
    '''

    # Skip over the header.
    line = gradefile.readline()
    while line != '\n':
        line = gradefile.readline()
```

```
    # Read the grades, accumulating them into a list.

    grades = []

    line = gradefile.readline()
    while line != '':
        # Now we have s string containing the info for a
        # single student.
        # Find the last space and take everything after that
        # space.
        grade = line[line.rfind(' ') + 1:]
        grades.append(float(grade))
        line = gradefile.readline()

    return grades

def count_grade_ranges(grades):
    '''(list of float) -> list of int

    Return a list of int where each index indicates how many grades were in these
    ranges:

    0-9: index 0
    10-19: 1
    20-29: 2
      :
    90-99: 9
    100:   10

    >>> count_grade_ranges([77.5, 37.5, 0.5, 9.5, 72.5, 100.0, 55.0, 70.0, 79.5])
    [2, 0, 0, 1, 0, 1, 0, 4, 0, 0, 1]
    '''

    range_counts = [0] * 11

    for grade in grades:
        which_range = int(grade // 10)
        range_counts[which_range] = range_counts[which_range] + 1

    return range_counts


def write_histogram(range_counts, histfile):
    '''(list of int, file open for writing) -> NoneType

    Write a histogram of *'s based on the number of grades in each range.

    The output format:
    0-9:   *
    10-19: **
    20-29: ******
      :
    90-99: **
    100:   *

    '''

    histfile.write('0-9:   ')
    histfile.write('*' * range_counts[0])
    histfile.write('\n')

    # Write the 2-digit ranges.
    for i in range(1, 10):
        low = i * 10
        high = i * 10 + 9
        histfile.write(str(low) + '-' + str(high) + ': ')
        histfile.write('*' * range_counts[i])
```

```
        histfile.write('\n')

    histfile.write('100:   ')
    histfile.write('*' * range_counts[-1])
    histfile.write('\n')
```

Grade histogram

```
import tkinter.filedialog
import grade

a1_filename = tkinter.filedialog.askopenfilename()
a1_file = open(a1_filename, 'r')

a1_histfilename = tkinter.filedialog.asksaveasfilename()
a1_histfile = open(a1_histfilename, 'w')

# Read the grades into a list.
grades = grade.read_grades(a1_file)

# Count the grades per range.
range_counts = grade.count_grade_ranges(grades)

# print(range_counts)

# Write the histogram to the file.
grade.write_histogram(range_counts, a1_histfile)

a1_file.close()
a1_histfile.close()
```

Optional reading

Optional reading

Optional reading

Optional reading