

# Lists and File Reading (Nested Loops)

# Check-in and Reminders

- Reminder: **pick up graded Homework 1** from up front
- Today in CS colloquium:
  - Thesis student talks
  - 2.35 pm in TCL 123 (Wege)
- Resources tab on course page:

[Typical workflows](#)

[Viewing Lab Grades in GitLab](#)

[Duane's Incredibly Brief Intro to Unix and Emacs](#)

[Python.org Python Tutorial](#)

[Python Standard Library](#)

[Python Language Reference](#)

[VPN Instructions for Accessing GitLab from off-campus](#)

**Do You Have Any Questions?**


# Last Class

- We learnt about sequences such as strings and lists
  - How their indexing works
- For loops!
  - Used when we have a known sequence that we want to iterate over
- While loops!
  - Used when we don't know stopping condition ahead of time
- Built a bunch of functions on sequences along the way
  - Do you remember any of them?


# Review: Syntax of Loops

```
while continuation_condition :  
    statement1  
    :  
    statementN
```

a boolean expression denoting whether to iterate through the body of the loop one more time.




A variable that takes its values from the items of the sequence.

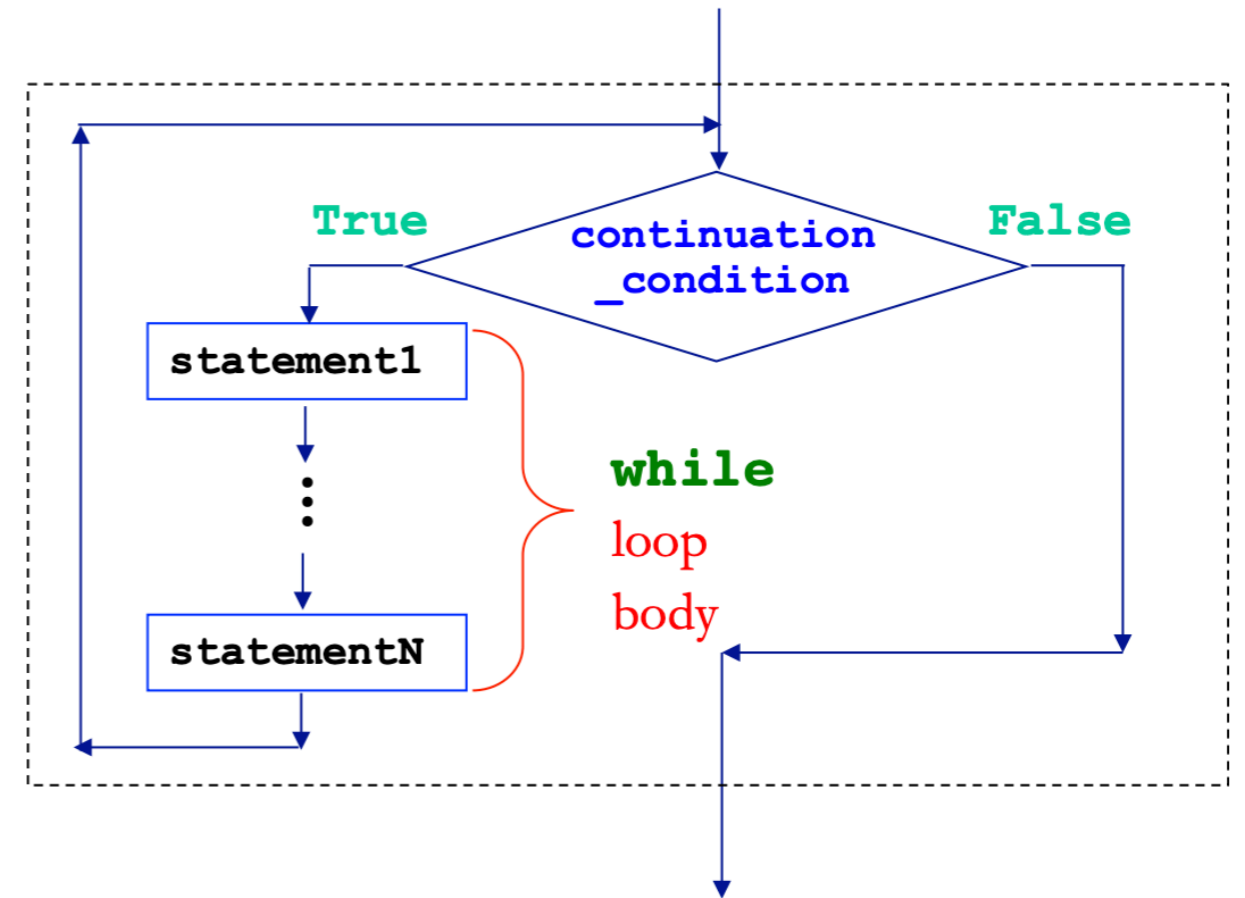
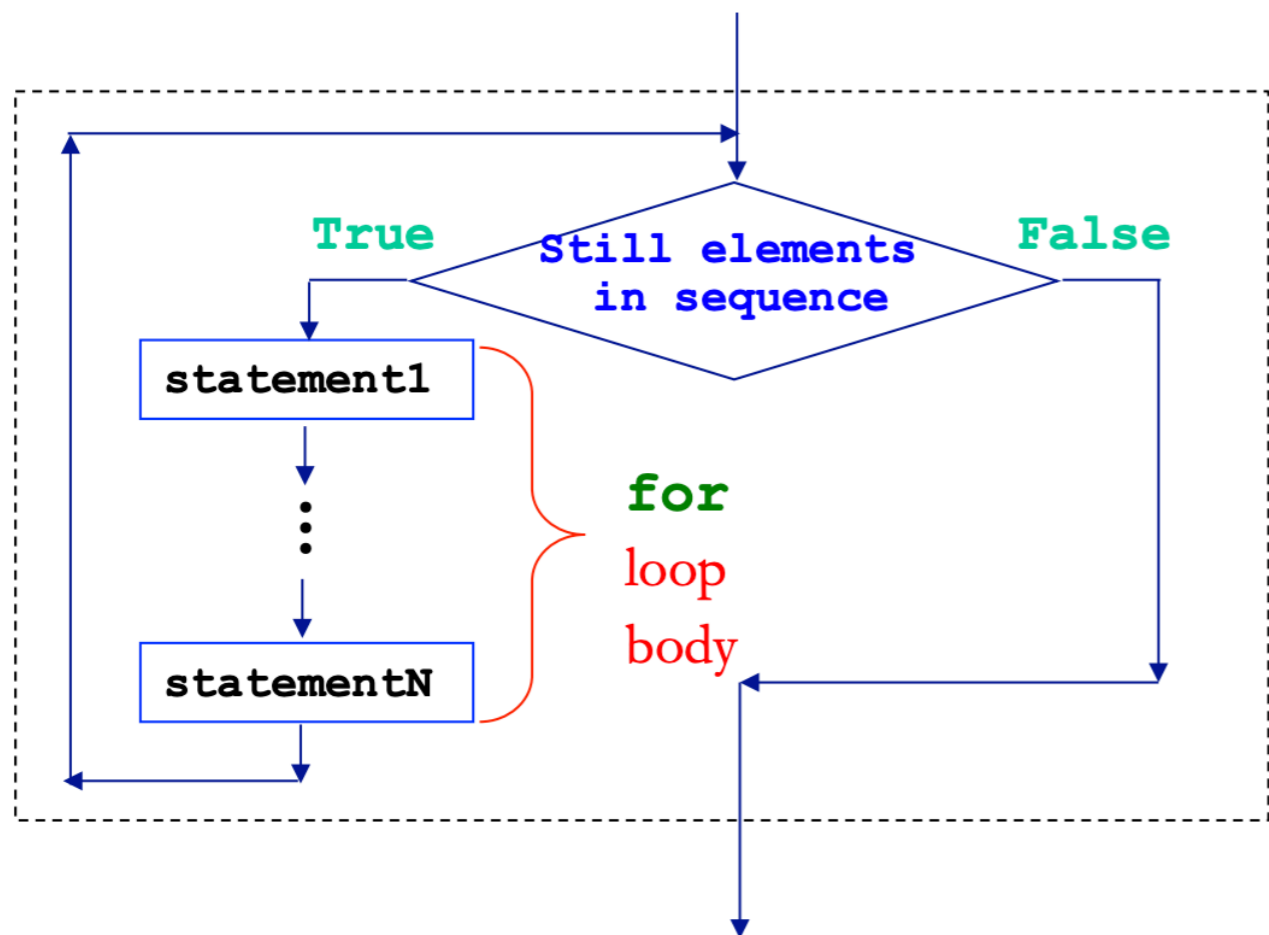


```
for var in sequence:  
    statement1  
    :  
    statementN
```

A sequence of items: characters in a string, items in a list, ranges, etc.



# Review: Loops as Flow Charts



# Today's Class

- Learning goals:
  - How to build, test, and use a module
  - How to test functions using doctests
  - `if `__name__` == “__main__”` block
  - Nested loops
  - How to read from a file
  - How to accumulate in a list
    - Concatenating lists
    - Appending to list

# Modules and Scripts

- **Script** is generally any piece of code saved in a file, e.g., `phase.py`
- Scripts are meant to be directly executed with: `python3 phase.py`
- A module are generally collection of statements and definitions (a sort of a library) that is meant to be imported and used by a different program
- Within a module, the module's name is available in a variable called `__name__`
- When a module is executed to be run directed as a script (as opposed to being imported), the `__name__` variable is set to `main`
- Why does this matter? **Importing a module runs it**, and we often want different behavior when the code is run as script vs when its imported as a module

# Importing a Module

## `__all__` special variable

- If the variable starts/ends with “\_ \_” it’s a special python variable
- We saw this with `__name__`
- `__all__` is another special variable
- Whatever is stored in `__all__` is imported when the user types:

```
from moduleName import *
```

- Any specific function/variable/etc. in the module can also be explicitly imported as:

```
from moduleName import explicitVariableName
```



```
if __name__ == '__main__'
```

- We can place code that we want to run when our module is executed as a script inside the `if __name__ == '__main__': block`
- This is usually testing code and we do not want run when we are importing functions from the file
- For example, all the definition functions we have design on sequences and loops are now in the file `sequenceTools.py`
- Notice the code at the bottom of the file under `if __name__ == '__main__': block`
  - This code block will be run when we execute `python3 sequenceTools.py`
  - This code block will not be run when we import functions from this module

# Testing Functions: Doctests

- Python's `doctest` module allows you to embed test cases and expected output directly into a functions `docstring`
- To use the `doctest` module we must import it using `import doctest`
- To make sure the test cases are run when the program is run as a script from the terminal, we need to call `doctest.testmod()`.
- To make sure that the tests are not run in an interactive shell or when the functions from the module are imported, we should place the command within a guarded `if __name__ == "__main__"`: block, e.g.

```
if __name__ == "__main__":  
    import doctest  
    doctest.testmod()
```

# List Accumulations

- It is often the case that we use loops to iterate over a sequence to "accumulate" certain items from it.
- Suppose someone gave us a list of words and we want to collect all words in that list that start with a vowel.
  - First we need to be able to iterate over all words in the master list
  - Then, for each word we must check if it starts with a vowel
  - If the word starts with a vowel, we want to store it somewhere
  - Since we want to store a collection of words, we can use a list type
- Such processes where we are accumulating something in a list are called \*list accumulation\*. You can accumulate items in a list using concatenation, similar to strings.

# List Accumulation Example

- Define a function `vowelList` that iterates over a given list of words `wordList` and collects all the words in the list that begin with a vowel (in a new list) and returns that list.

```
def vowelWordList(wordList):  
    '''Returns a list of words that start with a vowel from the input list'''  
    result = [] # initialize list accumulation variable  
    for word in wordList:  
        if startsWithVowel(word):  
            result.append(word)  
    return result
```

```
In [22]: phrase = ['The', 'sun', 'rises', 'in', 'the', \  
                  'east', 'and', 'sets', 'in', 'the', 'west']
```

```
In [23]: vowelWordsAccumulator(phrase)
```

```
Out[23]: ['in', 'east', 'and', 'in']
```

# A Nest Loop for Printing

A **for** loop body can contain a **for** loop.

```
Outer loop { for i in range(2, 6):  
              for j in range(2, 6):  
                  print(i, 'x', j, '=', i*j) } Inner loop
```

```
# print the multiplication table from 2 to 5
```

```
2 x 2 = 4  
2 x 3 = 6  
2 x 4 = 8  
2 x 5 = 10  
3 x 2 = 6  
3 x 3 = 9  
3 x 4 = 12  
...
```

To notice:

- Variable **i** in the outer loop is set initially to value 2.
  - Variable **j** in the inner loop is set initially to value 2.
  - Variable **j** keeps changing its value: 3, 4, 5; meanwhile **i** doesn't change.
- When the inner loop is done, **i** becomes 3.
  - Now the inner loop starts again, and **j** takes on the values 2, 3, 4, 5
- Every time **j** reaches 5, the inner loop ends and **i** increments.
- The outer loop ends when both **i** and **j** are 5.

# Reading Files: Open

- `open(filename, mode)` returns a file object
  - `filename` is a path to a file
  - `mode` is a string where
    - 'r' - open for reading (default)
    - We will only look at this mode today
- Technically when you open a file, you must also close it
- To avoid writing code to explicitly open and close, we will use the `with... as` block which keeps the file open within it
- Today's focus: file objects are **iterable**
  - We will see how to iterate over the lines of a file

# Reading Files: with .. as

with open(filename) as inputFile:

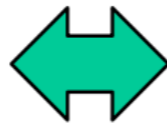
# do something with file

Path to file on computer as a string

Variable name for your file object

**Note. (syntax)** Indentation defines the body of the with block where the file is open

```
f = open(filename, 'r')  
... file operations involving f ...  
f.close()
```



```
with open(filename, 'r') as f:  
    ... file operations involving f ...  
    # f implicitly closed  
    # when with is done.
```

# Iterating over Lines in a File

- Within a `with open(filename) as inputFile:` block, we can iterate over the lines in the file just as we would iterate over any sequence such as lists or strings
- A line in the file is determined by the special newline character `'\n'`
- For us visually, a line has the regular meaning
- I have a text file called `classNames.txt` within a directory `textfiles`, so I would iterate and print each line in it as follows:

```
with open('textfiles/classNames.txt') as roster: # roster: name of file object
    for line in roster:
        print(line)
# file is implicitly closed here
```

Variable name for your file object

Path to file on computer as a string



# String Functions Helpful in File Reading

- When iterating over the lines of a file, the line variable will be a string ending in a special newline character `'\n'`
- Using the string function `line.strip()`: removes leading and trailing whitespace
- To break up a string of words (such as a line in a file) into a list of the constituent words, we can use `line.split()`: **.split** will split a string into a list based on a character (default is a **space**)
- Try these functions out in interactive python!

```
with open('textfiles/classNames.txt') as roster: # roster: name of file object
    for line in roster:
        print(line)
# file is implicitly closed here
```

line variable stores a string terminated by `'\n'`

# Class Coding Exercises

- Now that we know how to write nested loops, accumulate in lists and read from files, let us do some fun exercises with these concepts.
- We already built some helper functions in last class and today that play with sequences, we can use them to analyze files such as the book **Pride and Prejudice**
- We can ask questions such as:
  - How many words in Pride and Prejudice begin with a Vowel
  - How many words in Pride and Prejudice start and end with the same letter?
  - How many names are common between students in this class and Pride and Prejudice!
  - Anything else fun?

# Acknowledgments

These slides have been adapted from:

- <http://cs111.wellesley.edu/spring19> and
- <https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/>