

CS 115 Lecture 19

More about files; 2D lists

Neil Moore

Department of Computer Science
University of Kentucky
Lexington, Kentucky 40506
`neil@cs.uky.edu`

24 November 2015

Modifying files

We saw last time how to read and write from a file.

- What if we want to modify a file?

Modifying files

We saw last time how to read and write from a file.

- What if we want to modify a file?
 - ▶ Need to both read and write.
 - ▶ So we'll have to open the file twice.
 - ▶ But not at the same time!

Modifying files

We saw last time how to read and write from a file.

- What if we want to modify a file?
 - ▶ Need to both read and write.
 - ▶ So we'll have to open the file twice.
 - ▶ But not at the same time!
 - ▶ Because opening for writing truncates.

Modifying files

We saw last time how to read and write from a file.

- What if we want to modify a file?
 - ▶ Need to both read and write.
 - ▶ So we'll have to open the file twice.
 - ▶ But not at the same time!
 - ▶ Because opening for writing truncates.
- The idea:
 - ▶ Read in the whole file and close it.

Modifying files

We saw last time how to read and write from a file.

- What if we want to modify a file?
 - ▶ Need to both read and write.
 - ▶ So we'll have to open the file twice.
 - ▶ But not at the same time!
 - ▶ Because opening for writing truncates.
- The idea:
 - ▶ Read in the whole file and close it.
 - ▶ Process the contents.

Modifying files

We saw last time how to read and write from a file.

- What if we want to modify a file?
 - ▶ Need to both read and write.
 - ▶ So we'll have to open the file twice.
 - ▶ But not at the same time!
 - ▶ Because opening for writing truncates.
- The idea:
 - ▶ Read in the whole file and close it.
 - ▶ Process the contents.
 - ▶ Then open the file for writing.
 - ★ Using a *different* file object.

Modifying files

We saw last time how to read and write from a file.

- What if we want to modify a file?
 - ▶ Need to both read and write.
 - ▶ So we'll have to open the file twice.
 - ▶ But not at the same time!
 - ▶ Because opening for writing truncates.
- The idea:
 - ▶ Read in the whole file and close it.
 - ▶ Process the contents.
 - ▶ Then open the file for writing.
 - ★ Using a *different* file object.
 - ★ That might fail, even if reading worked!

Modifying files

We saw last time how to read and write from a file.

- What if we want to modify a file?
 - ▶ Need to both read and write.
 - ▶ So we'll have to open the file twice.
 - ▶ But not at the same time!
 - ▶ Because opening for writing truncates.
- The idea:
 - ▶ Read in the whole file and close it.
 - ▶ Process the contents.
 - ▶ Then open the file for writing.
 - ★ Using a *different* file object.
 - ★ That might fail, even if reading worked!
 - ▶ Write the processed content back to the file.

Modifying files

We saw last time how to read and write from a file.

- What if we want to modify a file?
 - ▶ Need to both read and write.
 - ▶ So we'll have to open the file twice.
 - ▶ But not at the same time!
 - ▶ Because opening for writing truncates.
- The idea:
 - ▶ Read in the whole file and close it.
 - ▶ Process the contents.
 - ▶ Then open the file for writing.
 - ★ Using a *different* file object.
 - ★ That might fail, even if reading worked!
 - ▶ Write the processed content back to the file.
 - ▶ Finally, close the output file object.

Modifying files

We saw last time how to read and write from a file.

- What if we want to modify a file?
 - ▶ Need to both read and write.
 - ▶ So we'll have to open the file twice.
 - ▶ But not at the same time!
 - ▶ Because opening for writing truncates.
- The idea:
 - ▶ Read in the whole file and close it.
 - ▶ Process the contents.
 - ▶ Then open the file for writing.
 - ★ Using a *different* file object.
 - ★ That might fail, even if reading worked!
 - ▶ Write the processed content back to the file.
 - ▶ Finally, close the output file object.
- Let's write a program to reverse the lines of a file.
 - ▶ `reverse-lines.py`

Modifying files

We saw last time how to read and write from a file.

- What if we want to modify a file?
 - ▶ Need to both read and write.
 - ▶ So we'll have to open the file twice.
 - ▶ But not at the same time!
 - ▶ Because opening for writing truncates.
- The idea:
 - ▶ Read in the whole file and close it.
 - ▶ Process the contents.
 - ▶ Then open the file for writing.
 - ★ Using a *different* file object.
 - ★ That might fail, even if reading worked!
 - ▶ Write the processed content back to the file.
 - ▶ Finally, close the output file object.
- Let's write a program to reverse the lines of a file.
 - ▶ `reverse-lines.py`

Example: letter count

Let's see a program to count letters in a file.

- 26 different letters, 26 different counters?

Example: letter count

Let's see a program to count letters in a file.

- 26 different letters, 26 different counters?
- We don't want to make 26 different variables

```
▶ if char == "A":  
    a_count += 1  
if char == "B":  
    ...
```

Example: letter count

Let's see a program to count letters in a file.

- 26 different letters, 26 different counters?
- We don't want to make 26 different variables
 - ▶

```
if char == "A":  
    a_count += 1  
if char == "B":  
    ...
```
 - ▶ Ugh!

Example: letter count

Let's see a program to count letters in a file.

- 26 different letters, 26 different counters?
- We don't want to make 26 different variables

- ▶ if char == "A":
 a_count += 1
 if char == "B":
 ...

- ▶ Ugh!

- Instead, we'll make a list of counters.

- ▶ To initialise:

- `counts = [0] * 26`

- ▶ A list with 26 elements, all zero.

Example: letter count

Let's see a program to count letters in a file.

- 26 different letters, 26 different counters?
- We don't want to make 26 different variables

- ▶ if char == "A":
 a_count += 1
 if char == "B":
 ...

- ▶ Ugh!

- Instead, we'll make a list of counters.

- ▶ To initialise:

- `counts = [0] * 26`

- ▶ A list with 26 elements, all zero.

- Read through each character of each line.

- ▶ Find and increment the corresponding counter.

Example: letter count

Let's see a program to count letters in a file.

- 26 different letters, 26 different counters?
- We don't want to make 26 different variables
 - ▶ `if char == "A":`
 `a_count += 1`
 `if char == "B":`
 `...`
 - ▶ Ugh!
- Instead, we'll make a list of counters.
 - ▶ To initialise:
 `counts = [0] * 26`
 - ▶ A list with 26 elements, all zero.
- Read through each character of each line.
 - ▶ Find and increment the corresponding counter.

Converting characters to numeric codes

Last time we heard about ASCII and Unicode.

- They assign a numeric code to each different character.

Converting characters to numeric codes

Last time we heard about ASCII and Unicode.

- They assign a numeric code to each different character.
- Python has functions to convert between characters and code.
- `ord` takes a character and returns its numeric code.

```
code = ord("A")
```

Converting characters to numeric codes

Last time we heard about ASCII and Unicode.

- They assign a numeric code to each different character.
- Python has functions to convert between characters and code.
- `ord` takes a character and returns its numeric code.

```
code = ord("A")
```

- ▶ Argument is a single character, returns an integer.

Converting characters to numeric codes

Last time we heard about ASCII and Unicode.

- They assign a numeric code to each different character.
- Python has functions to convert between characters and code.
- `ord` takes a character and returns its numeric code.

```
code = ord("A")
```

- ▶ Argument is a single character, returns an integer.

- `chr` takes a numeric code and returns the character.

```
char = chr(65)
```

Converting characters to numeric codes

Last time we heard about ASCII and Unicode.

- They assign a numeric code to each different character.
- Python has functions to convert between characters and code.
- `ord` takes a character and returns its numeric code.

```
code = ord("A")
```

- ▶ Argument is a single character, returns an integer.

- `chr` takes a numeric code and returns the character.

```
char = chr(65)
```

- ▶ Argument is an integer, returns a one-character string.

Converting characters to numeric codes

Last time we heard about ASCII and Unicode.

- They assign a numeric code to each different character.
- Python has functions to convert between characters and code.
- `ord` takes a character and returns its numeric code.

```
code = ord("A")
```

- ▶ Argument is a single character, returns an integer.

- `chr` takes a numeric code and returns the character.

```
char = chr(65)
```

- ▶ Argument is an integer, returns a one-character string.
- ▶ Codes below 32 are control characters (newline, tab, ...)

Converting characters to numeric codes

Last time we heard about ASCII and Unicode.

- They assign a numeric code to each different character.
- Python has functions to convert between characters and code.
- `ord` takes a character and returns its numeric code.

```
code = ord("A")
```

- ▶ Argument is a single character, returns an integer.

- `chr` takes a numeric code and returns the character.

```
char = chr(65)
```

- ▶ Argument is an integer, returns a one-character string.
- ▶ Codes below 32 are control characters (newline, tab, ...)

- We can use these to convert letters into a list index.
 - ▶ We want to put "A" at index zero, "B" at 1, etc.

Converting characters to numeric codes

Last time we heard about ASCII and Unicode.

- They assign a numeric code to each different character.
- Python has functions to convert between characters and code.
- `ord` takes a character and returns its numeric code.

```
code = ord("A")
```

- ▶ Argument is a single character, returns an integer.

- `chr` takes a numeric code and returns the character.

```
char = chr(65)
```

- ▶ Argument is an integer, returns a one-character string.
- ▶ Codes below 32 are control characters (newline, tab, ...)

- We can use these to convert letters into a list index.

- ▶ We want to put "A" at index zero, "B" at 1, etc.
- ▶ So the index is `ord(char) - ord("A")`

Converting characters to numeric codes

Last time we heard about ASCII and Unicode.

- They assign a numeric code to each different character.
- Python has functions to convert between characters and code.
- `ord` takes a character and returns its numeric code.

```
code = ord("A")
```

- ▶ Argument is a single character, returns an integer.

- `chr` takes a numeric code and returns the character.

```
char = chr(65)
```

- ▶ Argument is an integer, returns a one-character string.
- ▶ Codes below 32 are control characters (newline, tab, ...)

- We can use these to convert letters into a list index.

- ▶ We want to put "A" at index zero, "B" at 1, etc.
- ▶ So the index is `ord(char) - ord("A")`
- ▶ To convert back: `chr(i + ord("A"))`

Converting characters to numeric codes

Last time we heard about ASCII and Unicode.

- They assign a numeric code to each different character.
- Python has functions to convert between characters and code.
- `ord` takes a character and returns its numeric code.

```
code = ord("A")
```

- ▶ Argument is a single character, returns an integer.

- `chr` takes a numeric code and returns the character.

```
char = chr(65)
```

- ▶ Argument is an integer, returns a one-character string.
- ▶ Codes below 32 are control characters (newline, tab, ...)

- We can use these to convert letters into a list index.

- ▶ We want to put "A" at index zero, "B" at 1, etc.
- ▶ So the index is `ord(char) - ord("A")`
- ▶ To convert back: `chr(i + ord("A"))`

- `lettercount.py`

Converting characters to numeric codes

Last time we heard about ASCII and Unicode.

- They assign a numeric code to each different character.
- Python has functions to convert between characters and code.
- `ord` takes a character and returns its numeric code.

```
code = ord("A")
```

- ▶ Argument is a single character, returns an integer.

- `chr` takes a numeric code and returns the character.

```
char = chr(65)
```

- ▶ Argument is an integer, returns a one-character string.
- ▶ Codes below 32 are control characters (newline, tab, ...)

- We can use these to convert letters into a list index.

- ▶ We want to put "A" at index zero, "B" at 1, etc.
- ▶ So the index is `ord(char) - ord("A")`
- ▶ To convert back: `chr(i + ord("A"))`

- `lettercount.py`

Nested loops

Notice that we had a loop inside another loop:

- For each line in the file:
 - ▶ For each character in the line.

Nested loops

Notice that we had a loop inside another loop:

- For each line in the file:
 - ▶ For each character in the line.
- This is called a **nested** loop.

Nested loops

Notice that we had a loop inside another loop:

- For each line in the file:
 - ▶ For each character in the line.
- This is called a **nested** loop.
- Which of the two loops iterates more frequently?

Nested loops

Notice that we had a loop inside another loop:

- For each line in the file:
 - ▶ For each character in the line.
- This is called a **nested** loop.
- Which of the two loops iterates more frequently?
 - ▶ The inner loop.
 - ▶ Line 1 char 1, line 1 char 2, line 1 char 3, ...

Nested loops

Notice that we had a loop inside another loop:

- For each line in the file:
 - ▶ For each character in the line.
- This is called a **nested** loop.
- Which of the two loops iterates more frequently?
 - ▶ The inner loop.
 - ▶ Line 1 char 1, line 1 char 2, line 1 char 3, ...
 - ▶ Then line 2 char 1, line 2 char 2, ...

Nested loops

Notice that we had a loop inside another loop:

- For each line in the file:
 - ▶ For each character in the line.
- This is called a **nested** loop.
- Which of the two loops iterates more frequently?
 - ▶ The inner loop.
 - ▶ Line 1 char 1, line 1 char 2, line 1 char 3, ...
 - ▶ Then line 2 char 1, line 2 char 2, ...
 - ▶ Once the inner loop finishes, go to the next iteration of the outer loop.

Nested loops

Notice that we had a loop inside another loop:

- For each line in the file:
 - ▶ For each character in the line.
- This is called a **nested** loop.
- Which of the two loops iterates more frequently?
 - ▶ The inner loop.
 - ▶ Line 1 char 1, line 1 char 2, line 1 char 3, ...
 - ▶ Then line 2 char 1, line 2 char 2, ...
 - ▶ Once the inner loop finishes, go to the next iteration of the outer loop.
 - ▶ What if something should happen before/after each row?
 - ★ (Like printing a newline?)

Nested loops

Notice that we had a loop inside another loop:

- For each line in the file:
 - ▶ For each character in the line.
- This is called a **nested** loop.
- Which of the two loops iterates more frequently?
 - ▶ The inner loop.
 - ▶ Line 1 char 1, line 1 char 2, line 1 char 3, ...
 - ▶ Then line 2 char 1, line 2 char 2, ...
 - ▶ Once the inner loop finishes, go to the next iteration of the outer loop.
 - ▶ What if something should happen before/after each row?
 - ★ (Like printing a newline?)
 - ★ Put it inside the outer loop but not the inner.

Nested loops

Notice that we had a loop inside another loop:

- For each line in the file:
 - ▶ For each character in the line.
- This is called a **nested** loop.
- Which of the two loops iterates more frequently?
 - ▶ The inner loop.
 - ▶ Line 1 char 1, line 1 char 2, line 1 char 3, ...
 - ▶ Then line 2 char 1, line 2 char 2, ...
 - ▶ Once the inner loop finishes, go to the next iteration of the outer loop.
 - ▶ What if something should happen before/after each row?
 - ★ (Like printing a newline?)
 - ★ Put it inside the outer loop but not the inner.
- Let's use a nested loop to write a multiplication table.
 - ▶ `mult-table.py`

Nested loops

Notice that we had a loop inside another loop:

- For each line in the file:
 - ▶ For each character in the line.
- This is called a **nested** loop.
- Which of the two loops iterates more frequently?
 - ▶ The inner loop.
 - ▶ Line 1 char 1, line 1 char 2, line 1 char 3, ...
 - ▶ Then line 2 char 1, line 2 char 2, ...
 - ▶ Once the inner loop finishes, go to the next iteration of the outer loop.
 - ▶ What if something should happen before/after each row?
 - ★ (Like printing a newline?)
 - ★ Put it inside the outer loop but not the inner.
- Let's use a nested loop to write a multiplication table.
 - ▶ `mult-table.py`

Counting iterations

- How many times did we print a number?

Counting iterations

- How many times did we print a number?
 - ▶ 10 times in the first iteration.
 - ▶ 10 times in the second iteration.
 - ▶ And so on. . .

Counting iterations

- How many times did we print a number?
 - ▶ 10 times in the first iteration.
 - ▶ 10 times in the second iteration.
 - ▶ And so on. . .
 - ▶ $10 \times 10 = 100$ times altogether.

Counting iterations

- How many times did we print a number?
 - ▶ 10 times in the first iteration.
 - ▶ 10 times in the second iteration.
 - ▶ And so on. . .
 - ▶ $10 \times 10 = 100$ times altogether.
- When the inner loop's sequence is the same each time:
 - ▶ Total iterations = outer iterations \times inner iterations.

Counting iterations

- How many times did we print a number?
 - ▶ 10 times in the first iteration.
 - ▶ 10 times in the second iteration.
 - ▶ And so on. . .
 - ▶ $10 \times 10 = 100$ times altogether.
- When the inner loop's sequence is the same each time:
 - ▶ Total iterations = outer iterations \times inner iterations.
 - ▶ If it's not the same, add up all the inner iterations.

Counting iterations

- How many times did we print a number?
 - ▶ 10 times in the first iteration.
 - ▶ 10 times in the second iteration.
 - ▶ And so on. . .
 - ▶ $10 \times 10 = 100$ times altogether.
- When the inner loop's sequence is the same each time:
 - ▶ Total iterations = outer iterations \times inner iterations.
 - ▶ If it's not the same, add up all the inner iterations.

Two-dimensional lists

- Nested loops are particularly useful when we have nested lists.

Two-dimensional lists

- Nested loops are particularly useful when we have nested lists.
- That is, a list that contains other lists.
 - ▶ Also called a **two-dimensional list**.
- Why would we use a 2D list?

Two-dimensional lists

- Nested loops are particularly useful when we have nested lists.
- That is, a list that contains other lists.
 - ▶ Also called a **two-dimensional list**.
- Why would we use a 2D list?
 - ▶ For storing a table of values.
 - ★ Anything you would put in a spreadsheet.

Two-dimensional lists

- Nested loops are particularly useful when we have nested lists.
- That is, a list that contains other lists.
 - ▶ Also called a **two-dimensional list**.
- Why would we use a 2D list?
 - ▶ For storing a table of values.
 - ★ Anything you would put in a spreadsheet.
 - ★ Weather forecasts: row = city, column = days.
 - ★ Grade book: row = student, column = assignment.

Two-dimensional lists

- Nested loops are particularly useful when we have nested lists.
- That is, a list that contains other lists.
 - ▶ Also called a **two-dimensional list**.
- Why would we use a 2D list?
 - ▶ For storing a table of values.
 - ★ Anything you would put in a spreadsheet.
 - ★ Weather forecasts: row = city, column = days.
 - ★ Grade book: row = student, column = assignment.
 - ▶ Or a game board.
 - ★ For example, an 8×8 chess board.

Two-dimensional lists

- Nested loops are particularly useful when we have nested lists.
- That is, a list that contains other lists.
 - ▶ Also called a **two-dimensional list**.
- Why would we use a 2D list?
 - ▶ For storing a table of values.
 - ★ Anything you would put in a spreadsheet.
 - ★ Weather forecasts: row = city, column = days.
 - ★ Grade book: row = student, column = assignment.
 - ▶ Or a game board.
 - ★ For example, an 8×8 chess board.
 - ★ Make a list of 8 rows.
 - ★ Each row is a list of 8 squares.

Two-dimensional lists

- Nested loops are particularly useful when we have nested lists.
- That is, a list that contains other lists.
 - ▶ Also called a **two-dimensional list**.
- Why would we use a 2D list?
 - ▶ For storing a table of values.
 - ★ Anything you would put in a spreadsheet.
 - ★ Weather forecasts: row = city, column = days.
 - ★ Grade book: row = student, column = assignment.
 - ▶ Or a game board.
 - ★ For example, an 8×8 chess board.
 - ★ Make a list of 8 rows.
 - ★ Each row is a list of 8 squares.
 - ▶ Matrices (MA 322).
 - ★ Very useful for computer graphics and “big data”!

Two-dimensional lists

- Nested loops are particularly useful when we have nested lists.
- That is, a list that contains other lists.
 - ▶ Also called a **two-dimensional list**.
- Why would we use a 2D list?
 - ▶ For storing a table of values.
 - ★ Anything you would put in a spreadsheet.
 - ★ Weather forecasts: row = city, column = days.
 - ★ Grade book: row = student, column = assignment.
 - ▶ Or a game board.
 - ★ For example, an 8×8 chess board.
 - ★ Make a list of 8 rows.
 - ★ Each row is a list of 8 squares.
 - ▶ Matrices (MA 322).
 - ★ Very useful for computer graphics and “big data”!

Row-major and column-major

There are two ways to organize 2D lists:

- **Row-major:** table is a list of rows.
 - ▶ Each row is a list of entries, one per column.
 - ▶ Outer (major) loop for rows, inner loop for columns.
 - ▶ Row number comes first.

Row-major and column-major

There are two ways to organize 2D lists:

- **Row-major:** table is a list of rows.
 - ▶ Each row is a list of entries, one per column.
 - ▶ Outer (major) loop for rows, inner loop for columns.
 - ▶ Row number comes first.
- **Column-major:** table is a list of columns.
 - ▶ Each column is a list of entries, one per row.
 - ▶ Outer loop for columns, inner loop for rows.
 - ▶ Column number comes first.

Row-major and column-major

There are two ways to organize 2D lists:

- **Row-major:** table is a list of rows.
 - ▶ Each row is a list of entries, one per column.
 - ▶ Outer (major) loop for rows, inner loop for columns.
 - ▶ Row number comes first.
- **Column-major:** table is a list of columns.
 - ▶ Each column is a list of entries, one per row.
 - ▶ Outer loop for columns, inner loop for rows.
 - ▶ Column number comes first.
- Row-major is more common. Why?

Row-major and column-major

There are two ways to organize 2D lists:

- **Row-major:** table is a list of rows.
 - ▶ Each row is a list of entries, one per column.
 - ▶ Outer (major) loop for rows, inner loop for columns.
 - ▶ Row number comes first.
- **Column-major:** table is a list of columns.
 - ▶ Each column is a list of entries, one per row.
 - ▶ Outer loop for columns, inner loop for rows.
 - ▶ Column number comes first.
- Row-major is more common. Why?
 - ▶ How do we write English?

Row-major and column-major

There are two ways to organize 2D lists:

- **Row-major:** table is a list of rows.
 - ▶ Each row is a list of entries, one per column.
 - ▶ Outer (major) loop for rows, inner loop for columns.
 - ▶ Row number comes first.
- **Column-major:** table is a list of columns.
 - ▶ Each column is a list of entries, one per row.
 - ▶ Outer loop for columns, inner loop for rows.
 - ▶ Column number comes first.
- Row-major is more common. Why?
 - ▶ How do we write English?
 - ▶ Left-to-right (column in the inner loop)
 - ▶ Then top-to-bottom (row in the outer loop).

Row-major and column-major

There are two ways to organize 2D lists:

- **Row-major:** table is a list of rows.
 - ▶ Each row is a list of entries, one per column.
 - ▶ Outer (major) loop for rows, inner loop for columns.
 - ▶ Row number comes first.
- **Column-major:** table is a list of columns.
 - ▶ Each column is a list of entries, one per row.
 - ▶ Outer loop for columns, inner loop for rows.
 - ▶ Column number comes first.
- Row-major is more common. Why?
 - ▶ How do we write English?
 - ▶ Left-to-right (column in the inner loop)
 - ▶ Then top-to-bottom (row in the outer loop).
 - ▶ Printing/reading loops are a little simpler with row-major lists.

Row-major and column-major

There are two ways to organize 2D lists:

- **Row-major:** table is a list of rows.
 - ▶ Each row is a list of entries, one per column.
 - ▶ Outer (major) loop for rows, inner loop for columns.
 - ▶ Row number comes first.
- **Column-major:** table is a list of columns.
 - ▶ Each column is a list of entries, one per row.
 - ▶ Outer loop for columns, inner loop for rows.
 - ▶ Column number comes first.
- Row-major is more common. Why?
 - ▶ How do we write English?
 - ▶ Left-to-right (column in the inner loop)
 - ▶ Then top-to-bottom (row in the outer loop).
 - ▶ Printing/reading loops are a little simpler with row-major lists.
- We'll use row-major for the rest of the class.
 - ▶ Whichever you use, it's important to be consistent.

Row-major and column-major

There are two ways to organize 2D lists:

- **Row-major:** table is a list of rows.
 - ▶ Each row is a list of entries, one per column.
 - ▶ Outer (major) loop for rows, inner loop for columns.
 - ▶ Row number comes first.
- **Column-major:** table is a list of columns.
 - ▶ Each column is a list of entries, one per row.
 - ▶ Outer loop for columns, inner loop for rows.
 - ▶ Column number comes first.
- Row-major is more common. Why?
 - ▶ How do we write English?
 - ▶ Left-to-right (column in the inner loop)
 - ▶ Then top-to-bottom (row in the outer loop).
 - ▶ Printing/reading loops are a little simpler with row-major lists.
- We'll use row-major for the rest of the class.
 - ▶ Whichever you use, it's important to be consistent.

Creating 2D lists

There are (at least) two ways to make a 2D list:

- Hard-code it by putting lists inside a list:

```
table = [ [ 1, 2, 3 ], [ 4, 5, 6 ], [ 7, 8, 9 ] ]
```

Creating 2D lists

There are (at least) two ways to make a 2D list:

- Hard-code it by putting lists inside a list:

```
table = [ [ 1, 2, 3 ], [ 4, 5, 6 ], [ 7, 8, 9 ] ]
```

- ▶ Row 0 is [1, 2, 3], etc.

Creating 2D lists

There are (at least) two ways to make a 2D list:

- Hard-code it by putting lists inside a list:

```
table = [ [ 1, 2, 3 ], [ 4, 5, 6 ], [ 7, 8, 9 ] ]
```

- ▶ Row 0 is [1, 2, 3], etc.

- Or use a two-step process:

- First, create an empty outer list.

```
table = []
```

- ▶ This list will hold our rows.

Creating 2D lists

There are (at least) two ways to make a 2D list:

- Hard-code it by putting lists inside a list:

```
table = [ [ 1, 2, 3 ], [ 4, 5, 6 ], [ 7, 8, 9 ] ]
```

- ▶ Row 0 is [1, 2, 3], etc.

- Or use a two-step process:

- First, create an empty outer list.

```
table = []
```

- ▶ This list will hold our rows.

- Then, create the row lists and append them to the outer list.

Creating 2D lists

There are (at least) two ways to make a 2D list:

- Hard-code it by putting lists inside a list:

```
table = [ [ 1, 2, 3 ], [ 4, 5, 6 ], [ 7, 8, 9 ] ]
```

- ▶ Row 0 is [1, 2, 3], etc.

- Or use a two-step process:

- First, create an empty outer list.

```
table = []
```

- ▶ This list will hold our rows.

- Then, create the row lists and append them to the outer list.

```
for rownum in range(5): # five rows
```

```
    row = [0, 0, 0] # three columns
```

```
    table.append(row)
```

Creating 2D lists

There are (at least) two ways to make a 2D list:

- Hard-code it by putting lists inside a list:

```
table = [ [ 1, 2, 3 ], [ 4, 5, 6 ], [ 7, 8, 9 ] ]
```

- ▶ Row 0 is [1, 2, 3], etc.

- Or use a two-step process:

- First, create an empty outer list.

```
table = []
```

- ▶ This list will hold our rows.

- Then, create the row lists and append them to the outer list.

```
for rownum in range(5): # five rows
```

```
    row = [0, 0, 0] # three columns
```

```
    table.append(row)
```

- Why not this? `table = [[0, 0, 0]] * 5`

Creating 2D lists

There are (at least) two ways to make a 2D list:

- Hard-code it by putting lists inside a list:

```
table = [ [ 1, 2, 3 ], [ 4, 5, 6 ], [ 7, 8, 9 ] ]
```

- ▶ Row 0 is [1, 2, 3], etc.

- Or use a two-step process:

- First, create an empty outer list.

```
table = []
```

- ▶ This list will hold our rows.

- Then, create the row lists and append them to the outer list.

```
for rownum in range(5): # five rows
```

```
    row = [0, 0, 0] # three columns
```

```
    table.append(row)
```

- Why not this? `table = [[0, 0, 0]] * 5`

- ▶ That makes all the rows aliases of one another!

Creating 2D lists

There are (at least) two ways to make a 2D list:

- Hard-code it by putting lists inside a list:

```
table = [ [ 1, 2, 3 ], [ 4, 5, 6 ], [ 7, 8, 9 ] ]
```

- ▶ Row 0 is [1, 2, 3], etc.

- Or use a two-step process:

- First, create an empty outer list.

```
table = []
```

- ▶ This list will hold our rows.

- Then, create the row lists and append them to the outer list.

```
for rownum in range(5): # five rows
```

```
    row = [0, 0, 0] # three columns
```

```
    table.append(row)
```

- Why not this? `table = [[0, 0, 0]] * 5`

- ▶ That makes all the rows aliases of one another!

Accessing 2D lists

Let's say we have a list with five rows and three columns.

- How do we access the element in the second row, first column?

Accessing 2D lists

Let's say we have a list with five rows and three columns.

- How do we access the element in the second row, first column?
- How do we access the second row?

Accessing 2D lists

Let's say we have a list with five rows and three columns.

- How do we access the element in the second row, first column?
- How do we access the second row?
`row = table[1]`

Accessing 2D lists

Let's say we have a list with five rows and three columns.

- How do we access the element in the second row, first column?
- How do we access the second row?

```
row = table[1]
```

- ▶ What type is the second row?

Accessing 2D lists

Let's say we have a list with five rows and three columns.

- How do we access the element in the second row, first column?
- How do we access the second row?

```
row = table[1]
```

- ▶ What type is the second row?
- ▶ A list.

Accessing 2D lists

Let's say we have a list with five rows and three columns.

- How do we access the element in the second row, first column?
- How do we access the second row?

```
row = table[1]
```

- ▶ What type is the second row?
- ▶ A list.
- ▶ So we need index 0 in the second row:

```
elt = row[0]
```

Accessing 2D lists

Let's say we have a list with five rows and three columns.

- How do we access the element in the second row, first column?
- How do we access the second row?

```
row = table[1]
```

- ▶ What type is the second row?
- ▶ A list.
- ▶ So we need index 0 in the second row:

```
elt = row[0]
```

- ▶ Can combine the two steps:

```
elt = table[1][0]
```

Accessing 2D lists

Let's say we have a list with five rows and three columns.

- How do we access the element in the second row, first column?
- How do we access the second row?

```
row = table[1]
```

- ▶ What type is the second row?
- ▶ A list.
- ▶ So we need index 0 in the second row:

```
elt = row[0]
```

- ▶ Can combine the two steps:

```
elt = table[1][0]
```

- To access an element in a 2D list:

```
list2d[row][column]
```

Accessing 2D lists

Let's say we have a list with five rows and three columns.

- How do we access the element in the second row, first column?
- How do we access the second row?

```
row = table[1]
```

- ▶ What type is the second row?
- ▶ A list.
- ▶ So we need index 0 in the second row:

```
elt = row[0]
```

- ▶ Can combine the two steps:

```
elt = table[1][0]
```

- To access an element in a 2D list:

```
list2d[row][column]
```

- ▶ For **row-major** lists.
- ▶ If the list is column-major, put the column first.

Accessing 2D lists

Let's say we have a list with five rows and three columns.

- How do we access the element in the second row, first column?
- How do we access the second row?

```
row = table[1]
```

- ▶ What type is the second row?
- ▶ A list.
- ▶ So we need index 0 in the second row:

```
elt = row[0]
```

- ▶ Can combine the two steps:

```
elt = table[1][0]
```

- To access an element in a 2D list:

```
list2d[row][column]
```

- ▶ For **row-major** lists.
- ▶ If the list is column-major, put the column first.

Traversing a 2D list

- To iterate over the contents of a 2D list, we need a nested loop.

Traversing a 2D list

- To iterate over the contents of a 2D list, we need a nested loop.
 - ▶ Outer loop: for each row (**row-major**)
 - ▶ Inner loop: for each column in that row.

Traversing a 2D list

- To iterate over the contents of a 2D list, we need a nested loop.
 - ▶ Outer loop: for each row (**row-major**)
 - ▶ Inner loop: for each column in that row.
- If we only need the elements, not indices:

```
for row in table: # row is a list
    for elt in row:
        process the element
    finish the row
```

Traversing a 2D list

- To iterate over the contents of a 2D list, we need a nested loop.
 - ▶ Outer loop: for each row (**row-major**)
 - ▶ Inner loop: for each column in that row.

- If we only need the elements, not indices:

```
for row in table: # row is a list
    for elt in row:
        process the element
    finish the row
```

- If we do need indices, use a range instead:

```
for rowno in range(len(table)): # len = number of rows
    for colno in range(len(table[rowno])):
        process element table[rowno][colno]
    finish the row
```

Traversing a 2D list

- To iterate over the contents of a 2D list, we need a nested loop.
 - ▶ Outer loop: for each row (**row-major**)
 - ▶ Inner loop: for each column in that row.

- If we only need the elements, not indices:

```
for row in table: # row is a list
    for elt in row:
        process the element
    finish the row
```

- If we do need indices, use a range instead:

```
for rowno in range(len(table)): # len = number of rows
    for colno in range(len(table[rowno])):
        process element table[rowno][colno]
    finish the row
```

A large example: tic-tac-toe

Let's look at a somewhat large program using 2D arrays.

- We'll use a 2D array to represent a tic-tac-toe board.
- Each element in the array will be "X", "O", or a space.

A large example: tic-tac-toe

Let's look at a somewhat large program using 2D arrays.

- We'll use a 2D array to represent a tic-tac-toe board.
- Each element in the array will be "X", "O", or a space.
- Use loops to print the board and check whether someone won.

A large example: tic-tac-toe

Let's look at a somewhat large program using 2D arrays.

- We'll use a 2D array to represent a tic-tac-toe board.
- Each element in the array will be "X", "O", or a space.
- Use loops to print the board and check whether someone won.
- `tictactoe.py`