Today's lecture will wrap up our work with lists. We will also talk about how to handle working with files.

Nested lists

2

Last time, we saw that Python allows us to work with lists of elements of any type. The cool thing is that we can have lists of lists, also known as nested lists.

Unusual lists

- `bizarre_list = [47, 'cs51', ['Feta','Cheddar', 'Edam', 'Gouda']]`
- What would `len(bizarre_list)` return?
  - 3
- What would `'Feta' in bizarre_list` return?
  - `False`
- What would `['Feta','Cheddar', 'Edam', 'Gouda'] in bizarre_list` return?
  - `True`

`bizarre_list` →

element	47	'cs51'	['Feta','Cheddar', 'Edam', 'Gouda']
index	0	1	2

3

Let's assume we have this unusual list where elements are of different type. First one seems to be an int, second an str, and third a list!? What do you think the length of this list is? It's three, because there are three elements. Note that an element within the inner list, such as 'Feta' cannot be accessed directly. in only considers the whole element, that is the entire list for the third element.

Nested lists

- Nested lists are lists of lists. *If* inner lists have the same length, you can think of them as matrices. For example,
- `matrix = [[ 1, 2, 3], [ 4, 5, 6], [ 7, 8, 9], [10, 11, 12]] has 4 rows and 3 columns`

`matrix` →

element	[1, 2, 3]	[4, 5, 6]	[7, 8, 9]	[10, 11, 12]
index	0	1	2	3

- If `matrix[0]` returns `[1, 2, 3]` what would the following return?
- `matrix[1]`
  - returns `[4, 5, 6]`
- `matrix[2]`
  - returns `[7, 8, 9]`
- `matrix[3]`
  - returns `[10, 11, 12]`

4

When you have a list whose elements are lists themselves, then this list is known as a nested list. If the inner lists have the same length, you can think of them as matrices (note that they don't have to have the same length). For example, here is a nested list with four inner lists, each having three elements. If we index matrix between 0-3, we will get the respective inner list back. You can think of matrix as having 4 rows and 3 columns.

Nested lists

- Nested lists are lists of lists
- *If* inner lists have the same length, you can think of them as matrices. For example,
- `matrix = [[ 1, 2, 3], [ 4, 5, 6], [ 7, 8, 9], [10, 11, 12]]`

`matrix` →

element	`[1, 2, 3]`	`[4, 5, 6]`	`[7, 8, 9]`	`[10, 11, 12]`
index	0	1	2	3

- `If matrix[0][0]` returns 1 and `matrix[1][1]` returns 5, what do these return?
- `matrix[2][1]`
  - returns 8
- `matrix[3][2]`
  - returns 12
- `matrix[3][3]`
  - returns `Error`

5

In turn, to access elements within any of the inner lists, we have to use two square brackets. The first corresponds to the row and the second to the column.

Practice Time

- What would the following code print?

```
a_list = [[4, [True, False], 6, 8], [888, 999]]
if a_list[0][1][0]:
    print(a_list[1][0])
else:
    print(a_list[1][1])
```

6

What do you think the following block of code would print?

Answer

- What would the following code print?

```
a_list = [[4, [True, False], 6, 8], [888, 999]]
if a_list[0][1][0]:
    print(a_list[1][0])
else:
    print(a_list[1][1])
```

- It would print 888

7

It would print 888. We first go to the first element [4, [True, False], 6, 8] then within the second element [True, False] and then within the first, i.e. True. This the if statement is satisfied and we would print the contents of the first element in the second inner list [[888, 999]], i.e. 888

Practice Time

- Define a function `nested_total` that takes a list of lists of `ints` and returns the sum of all the values.
- For example:

```
lst = [[1, 2], [3], [4, 5, 6]]
sum = nested_total(lst)
print(sum)
```

- Would print 21

Time to write some Python code yourself.

Answer

- Define a function `nested_total` that takes a list of lists of `ints` and returns the sum of all the values.

```
def nested_total(lst):
    sum = 0
    for i in range(len(lst)):
        for j in range(len(lst[i])):
            sum += lst[i][j]
    return sum
```


There are multiple ways of going about it but here's a basic solution that iterates through each inner list and then within each inner list through all elements and adds them up to a running total

Practice Time

- Define a function `nested_avg` that takes a list of lists of `ints` and returns a list with each sublist averaged
- For example:

```
list = [[1, 2], [3], [4, 5, 6]]
lst_avg = nested_avg(lst)
print(lst_avg)
```

- Would print `[1.5, 3.0, 5.0]`

10

How about writing a function that returns a list with each element being the average for each inner list

Answer

- Define a function `nested_avg` that takes a list of lists of `ints` and returns a list with each sublist averaged

```
def nested_avg(lst):
    lst_avg = []
    for i in range(len(lst)):
        length = len(lst[i])
        sum = 0
        for j in range(length):
            sum += lst[i][j]
        lst_avg.append(sum/length)
    return lst_avg
```

11

Again, there are many ways of going about it but here's a basic idea (although verbose). Within our function, we make an empty list. Like before, we iterate through each inner list and each element but we keep resetting the sum per inner list to calculate its average and then append it to the list that we will return

List comprehension

12

We will next proceed with seeing a neat trick to reduce the verbosity of our code when working with lists.

Appending elements to a list

```
courses = ['csci051', 'phys042', 'csci054', 'engl019']
cs_courses = []
for course in courses:
    if 'csci' in course:
        cs_courses.append(course)
print(cs_courses)
```

Would print `['csci051', 'csci054']`

13

Let's say we have a list of course we are enrolled this semester and want to create a list only with the ones offered in the cs department. We could start with an empty list, loop through each course and if it contains the word csci append it to an empty list.

List comprehension

```
courses = ['csci051', 'phys042', 'csci054', 'engl019']
cs_courses = [course for course in courses if 'csci' in course]
print(cs_courses)
```

Would also print `['csci051', 'csci054']`

14

We could do the same using list comprehension.

List comprehension syntax

```
courses = ['csci051', 'phys042', 'csci054', 'engl019']
cs_courses = [course for course in courses if 'csci' in course]
print(cs_courses)
```

Would also print `['csci051', 'csci054']`

In general, the syntax is:

- `newlist = [`*`expression`*` for `*`item`*` in `*`iterable`*` if `*`condition`*` == True]`
- iterable objects: string, list, tuple, range, etc
- Using list comprehension, create a new list of all courses that are not csci051.
- `non_cs051 = [course for course in courses if course != 'csci051']`

15

The general syntax for list comprehension is `newlist = [`*`expression`*` for `*`item`*` in `*`iterable`*` if `*`condition`*` == True]`. Iterable objects are objects such as strings, lists, tuples, range etc that we can iterate through.

Practice Time

- Consider the following code. What will `new_lst` evaluate to?

```
new_lst = []
for x in range(10):
    if x % 2 == 0:
        new_lst.append(x**2)
```

- Use list comprehension to shorten it

16

Your turn, what does this piece of code do and how we can shorten this piece of code?

Answer

- Consider the following code. What will `new_lst` evaluate to?

```
new_lst = []
for x in range(10):
    if x % 2 == 0:
        new_lst.append(x**2)
```

- `new_lst` would be `[0, 4, 16, 36, 64]`
- Use list comprehension to shorten it
- `[x**2 for x in range(10) if x % 2 == 0]`

17

Did you get this?

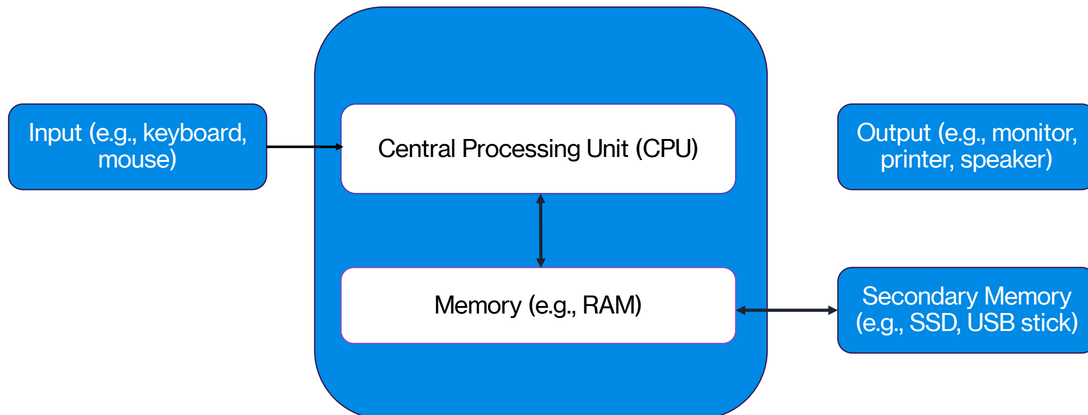
Files

18

We will next talk about how we process files

Files

- Programs are stored in main memory, i.e. RAM.
- Files are data stored in secondary memory, such as the hard drive, USB sticks. etc.



As we saw during the Systems unit in class, programs are stored in (main) memory, that is the RAM. That means all the variables they need to do their work are quickly accessible because RAM is fast. But once the program exits or RAM is not powered, we lose access to the program data. In contrast, if we want information to be permanently stored we would save it in files. Files are chunks of data in secondary memory, for example a solid state drive or a USB stick.

Files

- Files store data in the form of bytes.
- Although we make a distinction between text and binary files, text files are binary files that follow a specific encoding, most likely UTF-8, to interpret the bytes as valid characters (letters, numbers, symbols, emojis).

20

Files store data in the form of bytes. Although we make a distinction between text and binary files, text files are binary files that follow a specific encoding, most likely UTF-8, to interpret the bytes as valid characters (letters, numbers, symbols, emojis).

File system

- A **filesystem** is the way that an operating system organizes and provides access to store files.
- Files have a **filename** that we can identify them by.
- Files also reveal information about their type or appropriate applications to read them through their **extension**. E.g., .pdf shows that this is a pdf file and can be read with pdf readers such as Adobe Reader.
- They also contain **metadata** about their size, when they were created, last accessed, who created them or last modified them, who can read them/change them etc.
- Files are organized in **folders** or **directories** which themselves can have nested folders.
- We specify where a file resides through its **path**. For example, in my laptop, this presentation is in the directory /Users/apaa2017/Documents/cs051/lectures

21

The operating system provides a service called the filesystem that dictates how files are stored, organized, and accessed. A file is identified by its name. Its extension reveals its type and appropriate application to open it so that it is not read as raw binary data. Files also contain information like their size etc in metadata. They are organized in directories and their position is determined through a path.

Reading text files line by line

- To read a file line by line, we use the following template:

```
reader = open('filename.txt', 'r')
for line in reader:
        # do something
reader.close()
```

- `reader` is a reference to a file object.
- `filename.txt` gets replaced with the actual name of the file we want to read.
- The parameter `r` stands for read. That means we want to open the file to read it.
- The for in loop allows us to go line by line in the file. We could have done for example `print(line)`
- Don't forget to close the once you open it.

22

We will only focus on text files here. Python provides a very concise way of opening a file to read it. Once we open a file, we can read it line by line with a simple for loop. Don't forget to close a file reader once you open it.

Reading files as a list of strings

- Python also supports reading all lines of a file into a list of strings.
- Each string in the list represents a line from the file, and it includes the newline character ('\n') at the end of each line.
- ```
  reader = open('filename.txt', 'r')
  list_of_lines = reader.readlines()
  ```

23

Alternatively, we can also read all lines into a list of strings, with each string representing a file line.
  ```

Reading only a few lines of a file

- Python also supports reading one line at a time without using a for-each loop. For example, if you want to read the first 5 lines in a file you could do:

```
reader = open('filename.txt', 'r')
for i in range(10):
      reader.readline()
reader.close()
```

24

You can also read only a specified number of lines using a for i in range loop and calling the readline() method. Note that the file handler will remember that it stopped reading at the 10th line and if you try to read again it will continue at the 11th line.

(Over) Writing to files

- To write in a file, we use the following template:

```
writer = open('filename.txt', 'w')
writer.write('First line\n')
writer.write('Second line\n')
writer.close()
```

- `writer` is a reference to a file object.
- `filename.txt` gets replaced with the actual name of the file we want to write in. If it already exists, it overwrites it. If the file does not exist, creates a new file for writing.
- The parameter `w` stands for write. That means we want to open the file to write in it.
- We use the `write` method to write in the file object.
- Don't forget to close a file once you open it.

25

To write to a file, we open it and pass the 'w' parameter. Be mindful that if there is already a file with this name, its contents will be replaced that is overwritten this way. Note that the write method does not put new lines by default. You would need the '\n' special character.

(Appending when) Writing to files

- To write at the end of a file, we use the following template:

```
writer = open('filename.txt', 'a')
writer.write('Append to the end of the file\n')
writer.close()
```

- `writer` is a reference to a file object.
- `filename.txt` gets replaced with the actual name of the file we want to write in. If it already exists, the file pointer is at the end of the file. If the file does not exist, creates a new file for writing.
- The parameter *a* stands for append. That means we want to open the file to write at the end of it so that we don't lose existing content.
- We use the `write` method to write in the file object.
- Don't forget to close a file once you open it.

26

If you want to maintain the original file and append to it, you should use the 'a' option instead.

Practice Time

- Define a function `combine_files` that takes three parameters (`infile1, infile2, outfile`), all of which are strings and creates a new file named `outfile` whose contents are the contents of the file `infile1` followed by the contents of the file `infile2`.

Practice Time

```
def combine_files(infile1, infile2, outfile):
    f1 = open(infile1, 'r')
    f2 = open(infile2, 'r')
    out = open(outfile, 'w')
    for line in f1:
      out.write(line)
    for line in f2:
      out.write(line)
    f1.close()
    f2.close()
    out.close()
```

Practice Time

- Define a function `count_capital_letters` that takes on parameter `filename` and returns the number of capital letters in that file.
- You may use the `isupper` method to test if a character is uppercase.

Your time to write a function that counts the number of capital letters in a provided file.

Answer

- Define a function `count_capital_letters` that takes on parameter `filename` and returns the number of capital letters in that file.

```
def count_capital_letters(filename):
    count = 0
    opener = open(filename, 'r')
    for line in opener:
         for char in line:
             if char.isupper():
                 count += 1
    opener.close()
    return count
```

Practice Time

- Define a function `unique_words` that takes on parameter `filename` and returns a list with all the words in the file. If a word exists multiple times, it should only include it once in the list. You can split words by whitespace and ignore capitalization (i.e. use the method `lower`)

Answer

```
def unique_words(filename):
    words = []
    reader = open(filename, 'r')
    for line in reader:
        for word in line.split():
            clean_word = word.strip().lower()
            if clean_word not in words:
                words.append(clean_word)
    reader.close()
    return words
```

Errors

33

And to finish, we will talk about errors.

Errors

- You already have experience with things going wrong when programming. Maybe:
  - you forgot to pass an argument
  - tried to print an int
  - tried to access a list item out of the valid range of indices
- When working with files, you may also face trouble:
  - try to read a file in the wrong directory
  - try to write in a file without having permission to do so

Exceptions

- Exceptions are errors that we know how to handle.
- The syntax is

```
try:
    # some potentially dangerous code
except:
    # handle the error
finally:
    # code you want to execute no matter what
```

35

When there are errors like this that we know how to handle or at least we want to warn the user with a bit more information, we can use exceptions. To handle exceptions, we use the try except statement. In the try block, we put the code for which we know that things may go poorly. In the except block we handle the error. Optionally, we can also specify a finally block where we have code that should be executed no matter what.

Exceptions

- For example, when working with files

```
reader = open('filename.txt', 'r')
try:
  for line in reader:
       # do something
except:
    print('There was an error with the file when reading it line by line')
finally:
   reader.close()
```

36

For example, we may know that something might go wrong as we read a file line by line.

Raising Exceptions

- You can use the raise keyword to throw your own exceptions. For example,
- `raise Exception('CS51 Exception)`
- `raise ValueError('Invalid value')`

37

You can also throw your own exceptions using the raise keyword

Practice Time

- Define a function `pos_int_one_try` that asks the user to enter a positive integer. If the user enters a positive integer, the function returns that integer. If the user does not enter a positive integer, the function raises a `ValueError`.

- Hint: you can use the string method `isdigit()`

Answer

- Define a function `pos_int_one_try` that asks the user to enter a positive integer. If the user enters a positive integer, the function returns that integer. If the user does not enter a positive integer, the function raises a `ValueError`.

```
def return_pos_int_onetry():
    user_input = input('Enter a positive integer: ')
    if not user_input.isdigit() or int(user_input) < 0:
        raise ValueError(user_input + ' is not a positive integer.')
    return int(user_input)
```