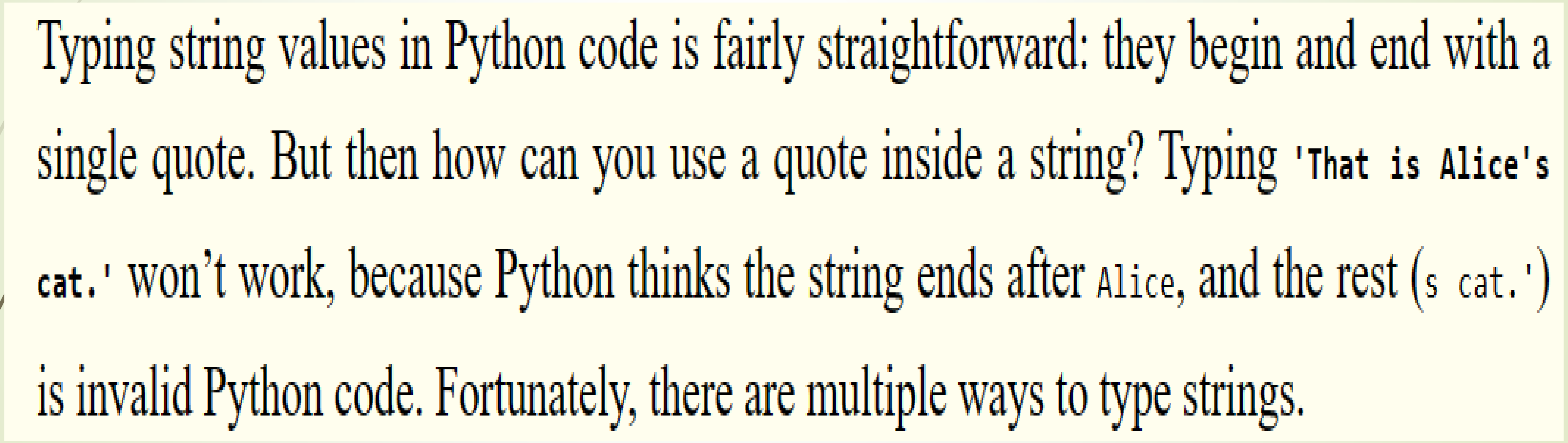
Module-3

A. Manipulating Strings

Dr.Sandeep Bhat
CSE,Dept.SIT Mangaluru

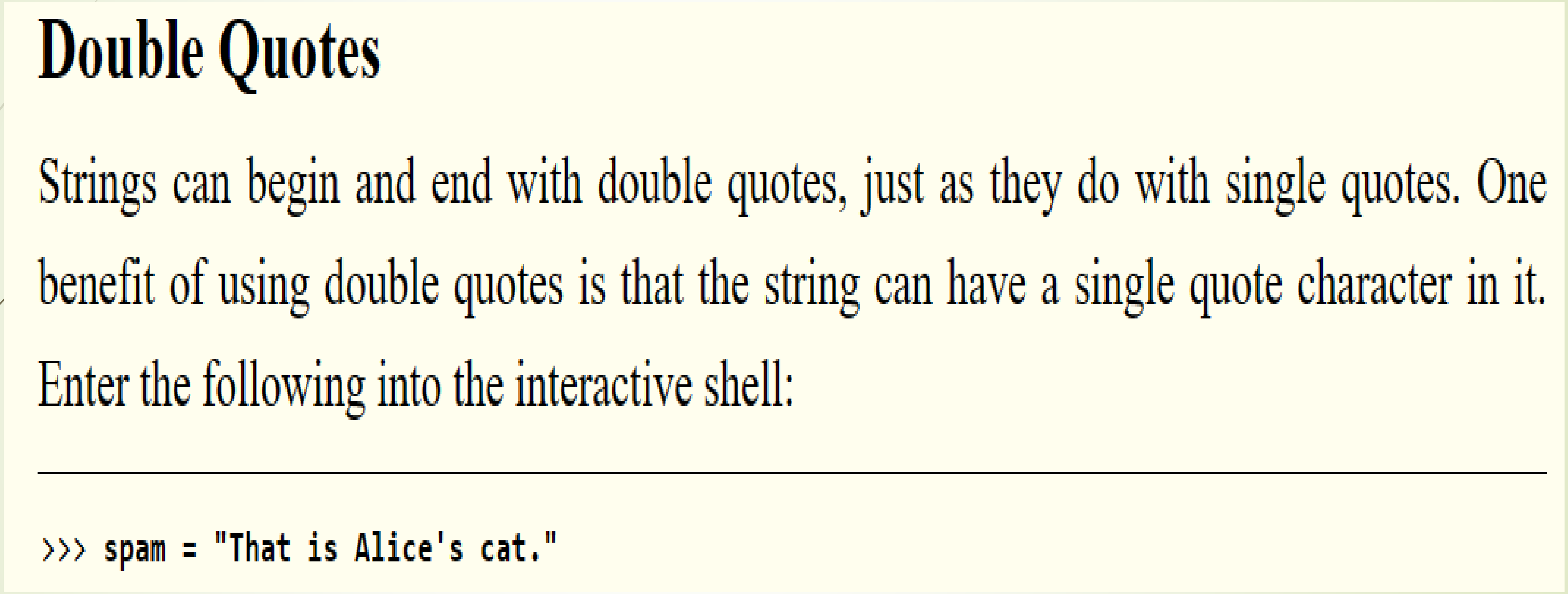
Working with Strings

String Literals



Typing string values in Python code is fairly straightforward: they begin and end with a single quote. But then how can you use a quote inside a string? Typing `'That is Alice's cat.'` won't work, because Python thinks the string ends after `Alice`, and the rest (`s cat.'`) is invalid Python code. Fortunately, there are multiple ways to type strings.

Cont..

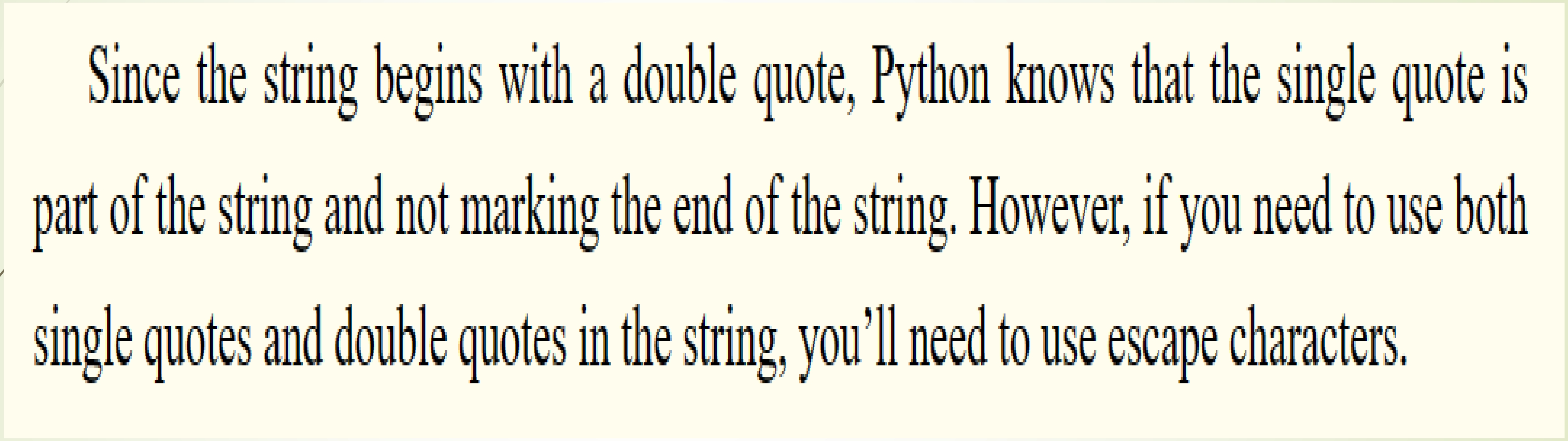


Double Quotes

Strings can begin and end with double quotes, just as they do with single quotes. One benefit of using double quotes is that the string can have a single quote character in it. Enter the following into the interactive shell:

```
>>> spam = "That is Alice's cat."
```

Cont..



Since the string begins with a double quote, Python knows that the single quote is part of the string and not marking the end of the string. However, if you need to use both single quotes and double quotes in the string, you'll need to use escape characters.

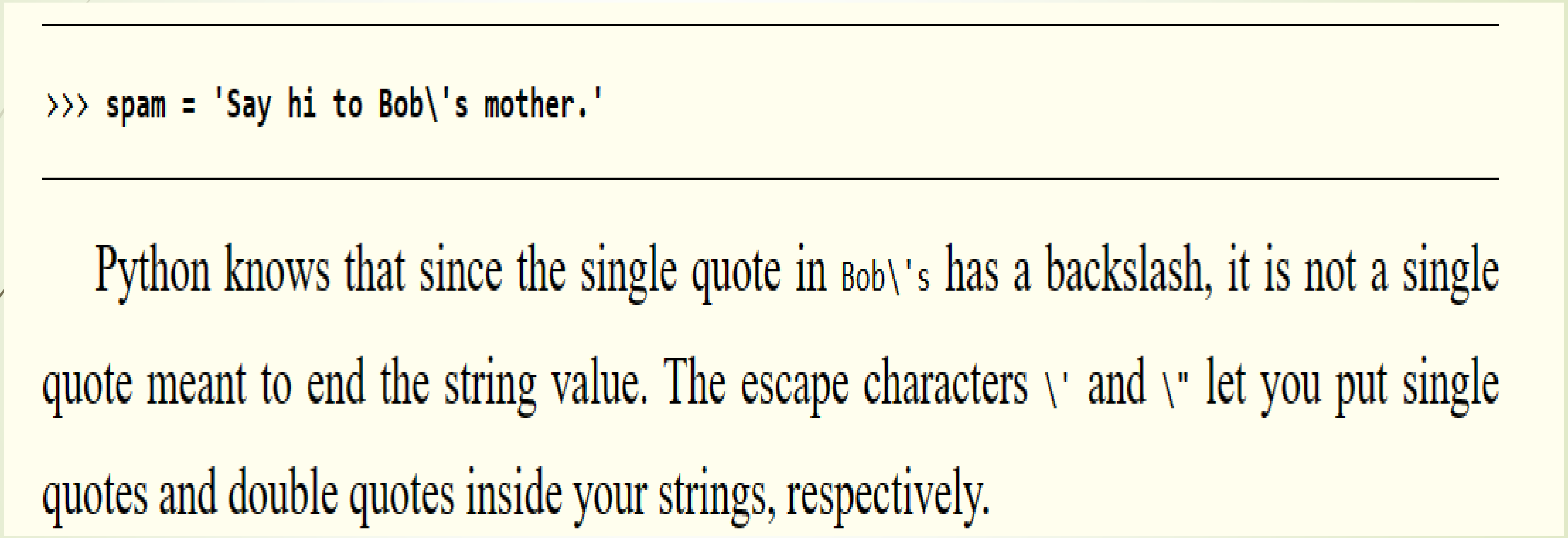
Escape Characters

- An *escape character* lets you use characters that are otherwise impossible to put into a string.

- An escape character consists of a backslash (\) followed by the character you want to add to the string. (Despite consisting of two characters, it is commonly referred to as a singular escape character.)

- For example, the escape character for a single quote is \'. You can use this inside a string that begins and ends with single quotes.

➢ To see how escape characters work, enter the following into the interactive shell:



```
>>> spam = 'Say hi to Bob\'s mother.'
```

Python knows that since the single quote in `Bob\'s` has a backslash, it is not a single quote meant to end the string value. The escape characters `\'` and `\"` let you put single quotes and double quotes inside your strings, respectively.

Escape Characters

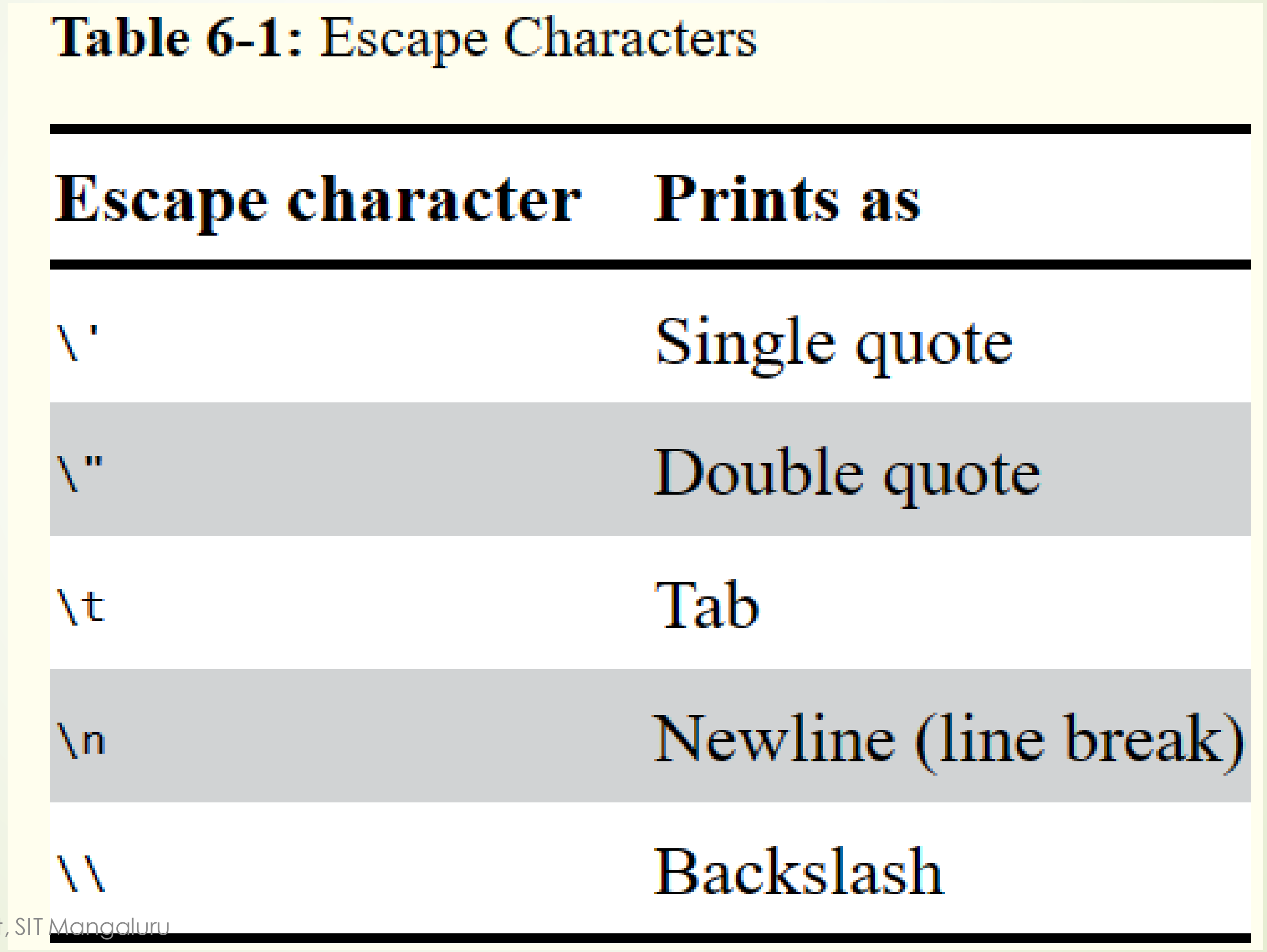
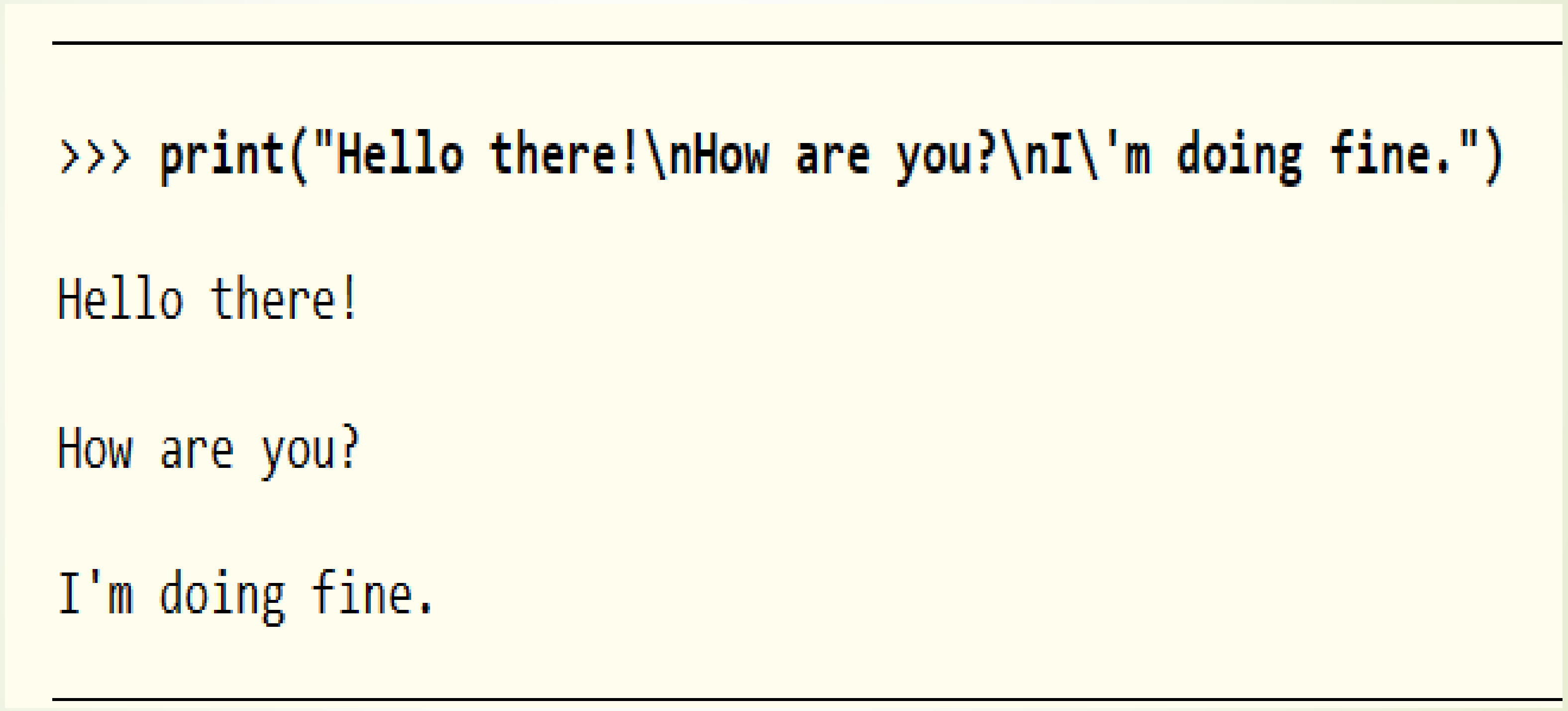


Table 6-1: Escape Characters

Escape character	Prints as
`\'`	Single quote
`\"`	Double quote
`\t`	Tab
`\n`	Newline (line break)
`\\`	Backslash

Cont..



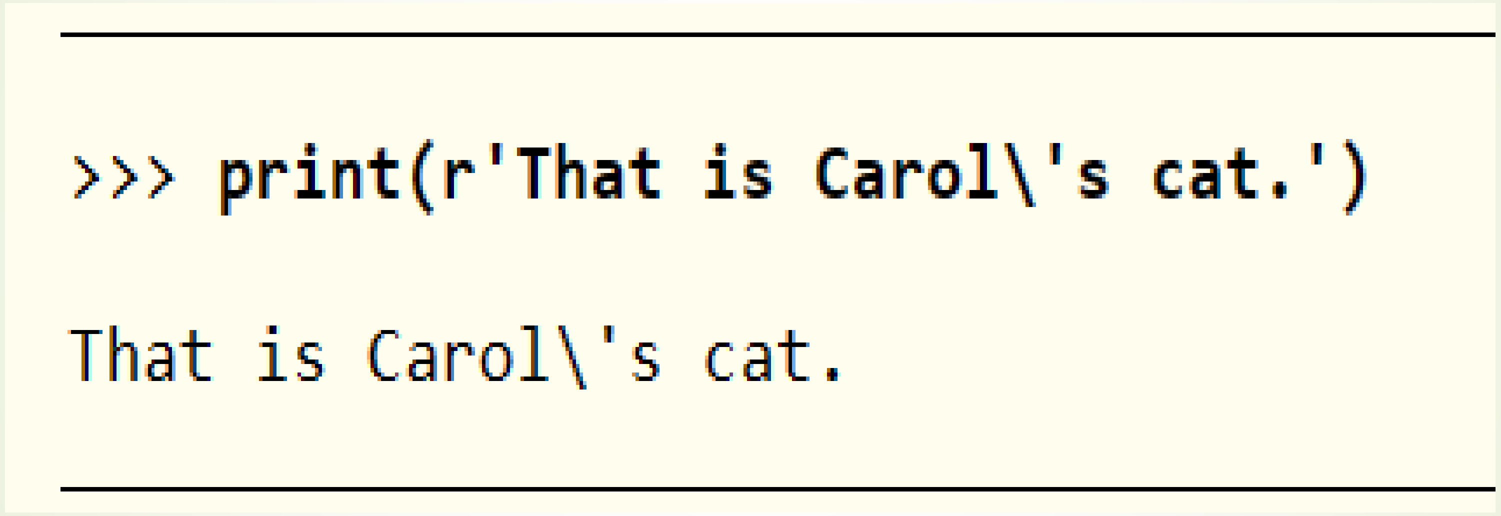
```
>>> print("Hello there!\nHow are you?\nI\'m doing fine.")
Hello there!
How are you?
I'm doing fine.
```

Raw Strings

- You can place an r before the beginning quotation mark of a string to make it a raw string.

- A *raw string* completely ignores all escape characters and prints any backslash that appears in the string.

Eg.



```
>>> print(r'That is Carol\'s cat.')
That is Carol\'s cat.
```

Cont..

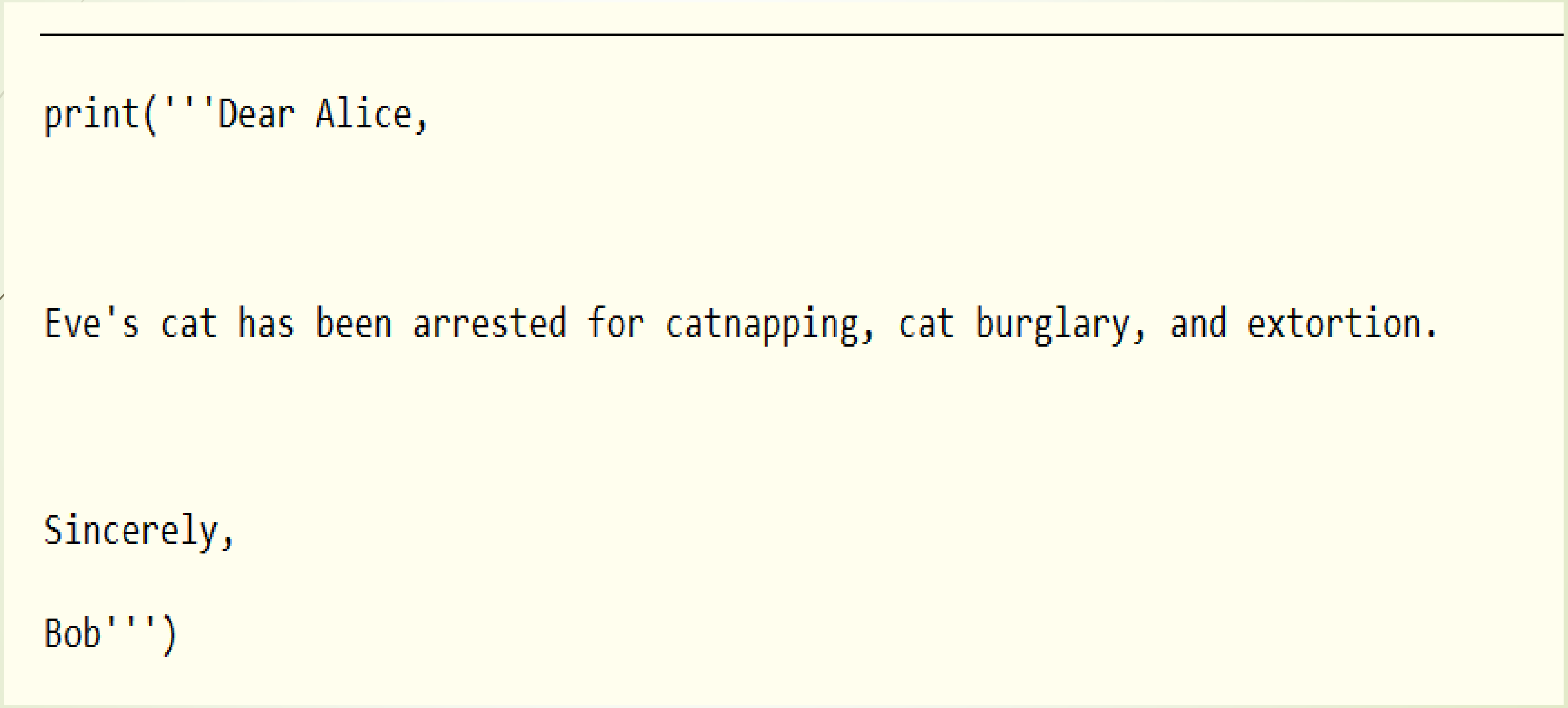
- Because this is a raw string, Python considers the backslash as part of the string and not as the start of an escape character.

- Raw strings are helpful if you are typing string values that contain many backslashes, such as the strings used for Windows file paths.

Multiline Strings with Triple Quotes

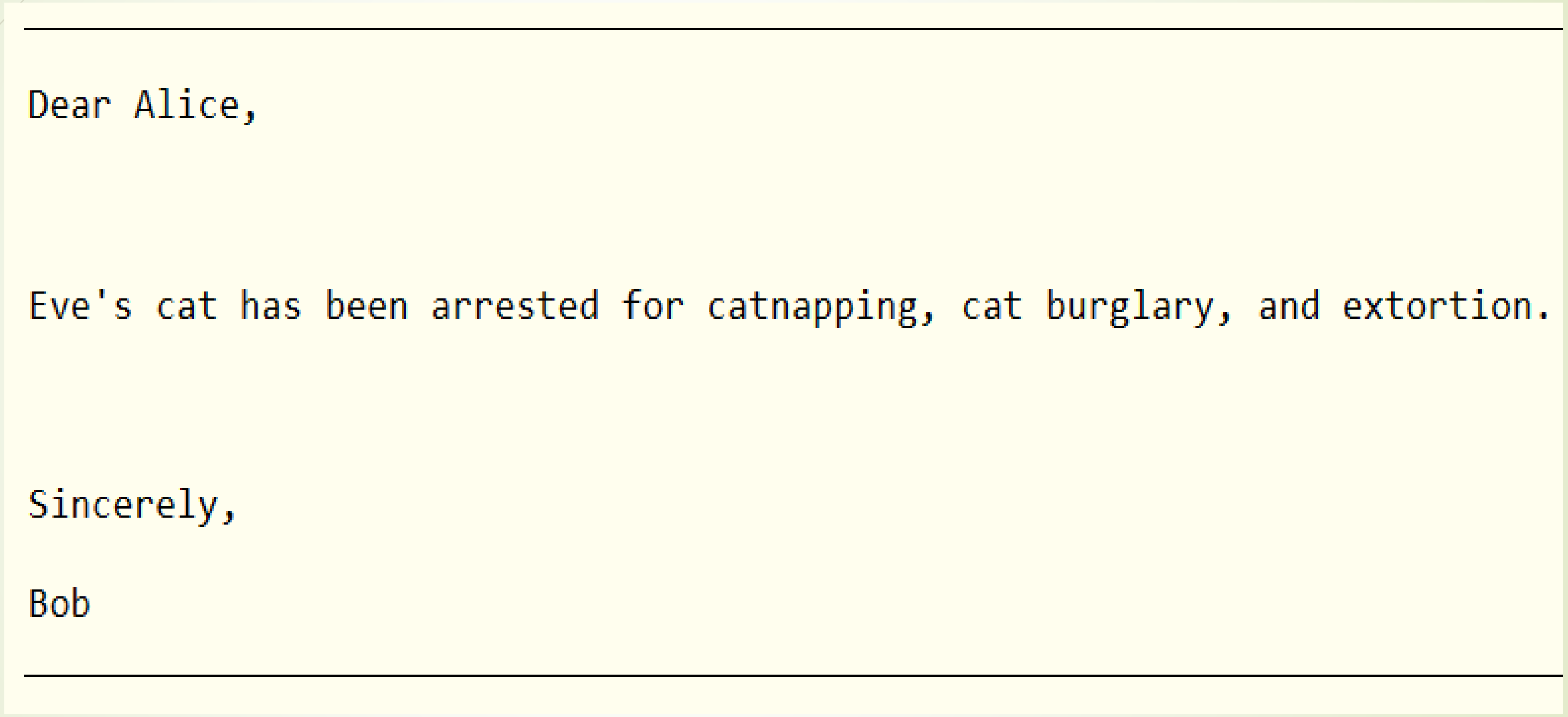
- While you can use the \n escape character to put a newline into a string, it is often easier to use multiline strings.
- A multiline string in Python begins and ends with either three single quotes or three double quotes.
- Any quotes, tabs, or newlines in between the "triple quotes" are considered part of the string.
- Python's indentation rules for blocks do not apply to lines inside a multiline string.

cont..



```
print('''Dear Alice,

Eve's cat has been arrested for catnapping, cat burglary, and extortion.

Sincerely,
Bob''')
```

output



```
Dear Alice,


Eve's cat has been arrested for catnapping, cat burglary, and extortion.


Sincerely,

Bob
```

Multiline Comments

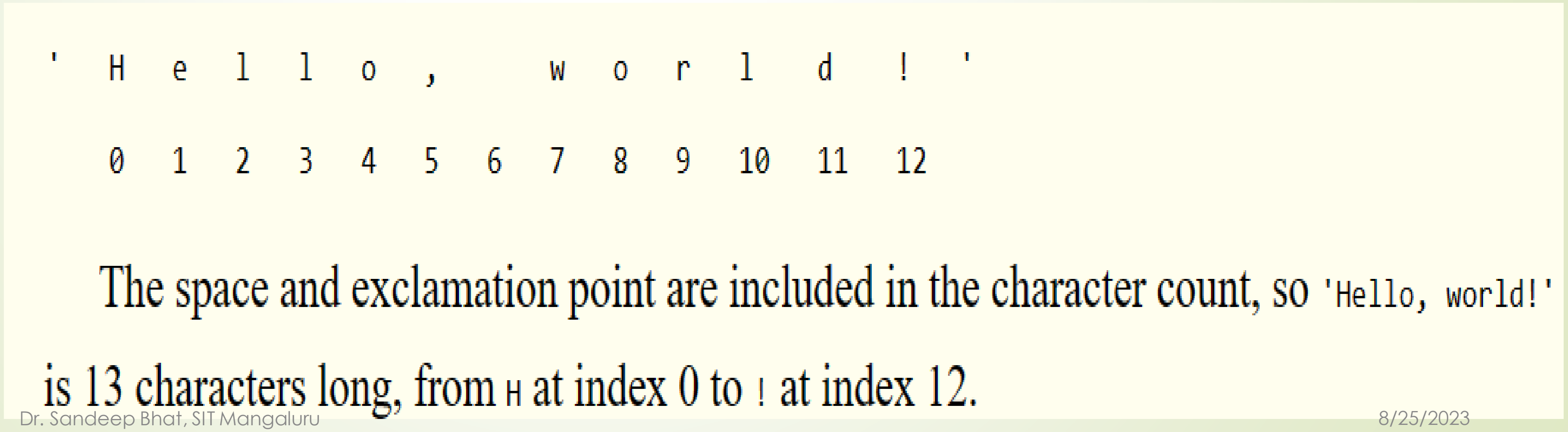
- While the hash character (#) marks the beginning of a comment for the rest of the line, a multiline string is often used for comments that span multiple lines.

Cont..

```
"""This is a test Python program.
Written by Al Sweigart al@inventwithpython.com

This program was designed for Python 3, not Python 2.
"""

def spam():
    """This is a multiline comment to help
    explain what the spam() function does."""
    print('Hello!')
```

Indexing and Slicing Strings

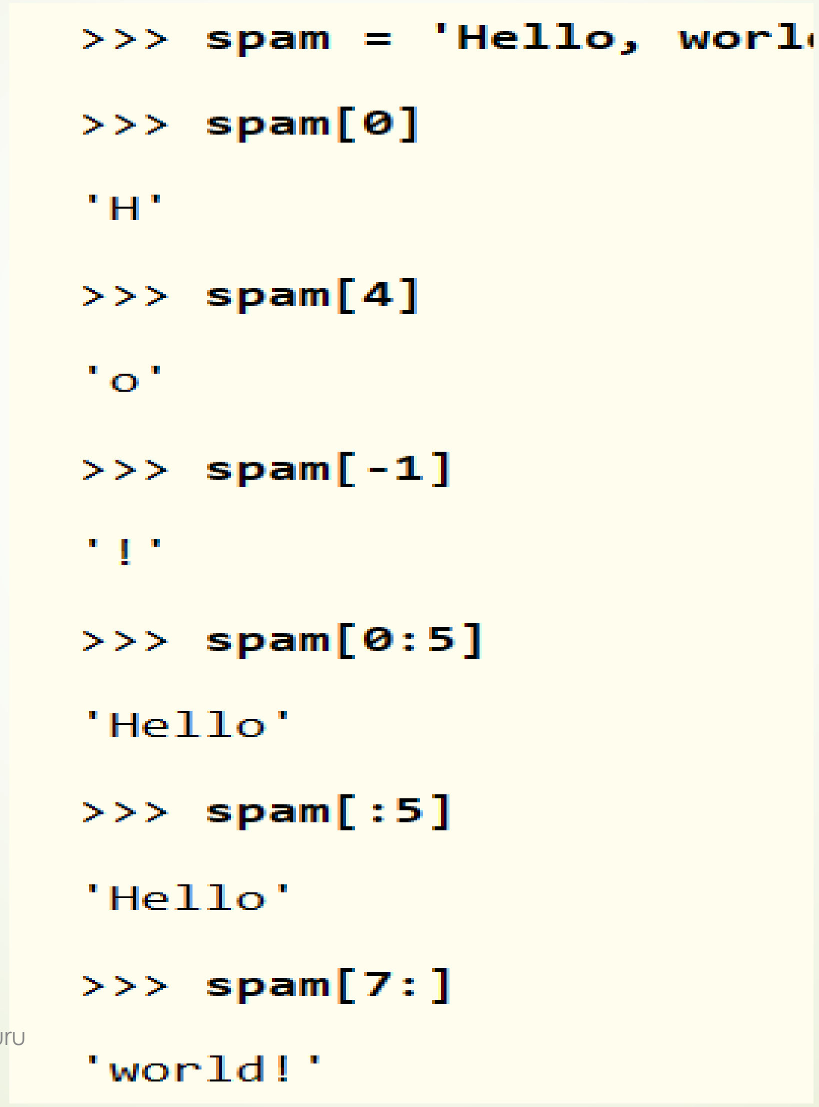
- Strings use indexes and slices the same way lists do.
- You can think of the string 'Hello, world!' as a list and each character in the string as an item with a corresponding index.



```
'  H  e  l  l  o  ,     w  o  r  l  d  !  '
   0  1  2  3  4  5  6  7  8  9  10 11 12
```

The space and exclamation point are included in the character count, so `'Hello, world!'` is 13 characters long, from `H` at index 0 to `!` at index 12.

Eg.

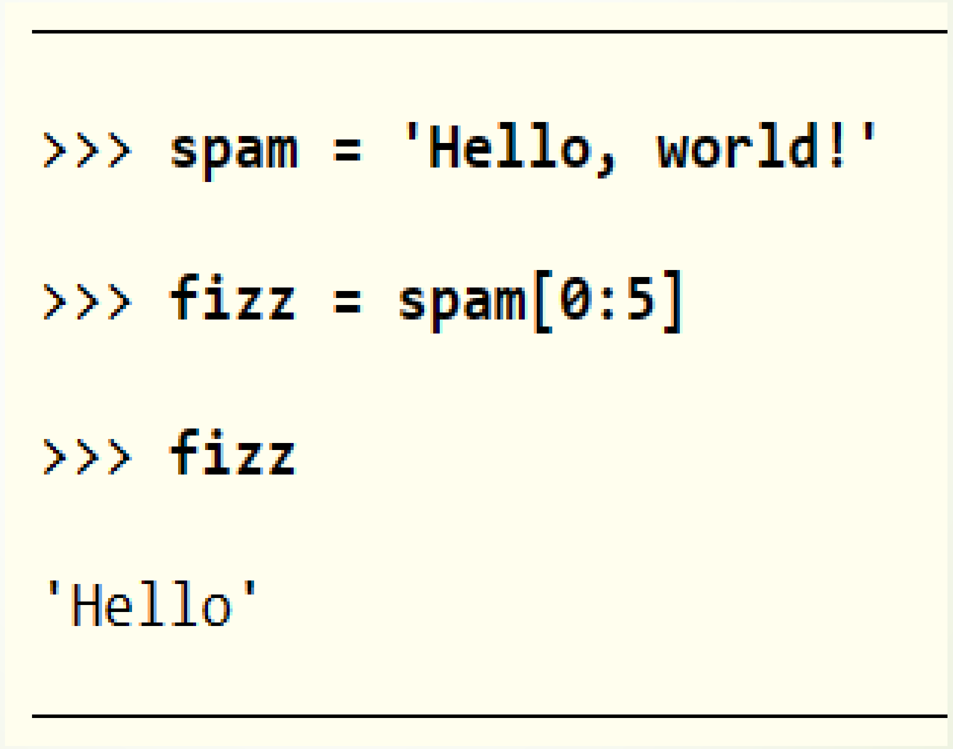


```
>>> spam = 'Hello, worl
>>> spam[0]
'H'
>>> spam[4]
'o'
>>> spam[-1]
'!'
>>> spam[0:5]
'Hello'
>>> spam[:5]
'Hello'
>>> spam[7:]
'world!'
```

Cont..

- If you specify an index, you’ll get the character at that position in the string.
- If you specify a range from one index to another, the starting index is included and the ending index is not.
- That’s why, if spam is 'Hello, world!', spam[0:5] is 'Hello'.
- The substring you get from spam[0:5] will include everything from spam[0] to spam[4], leaving out the comma at index 5 and the space at index 6.
- This is similar to how range(5) will cause a for loop to iterate up to, but not including, 5.

Eg.



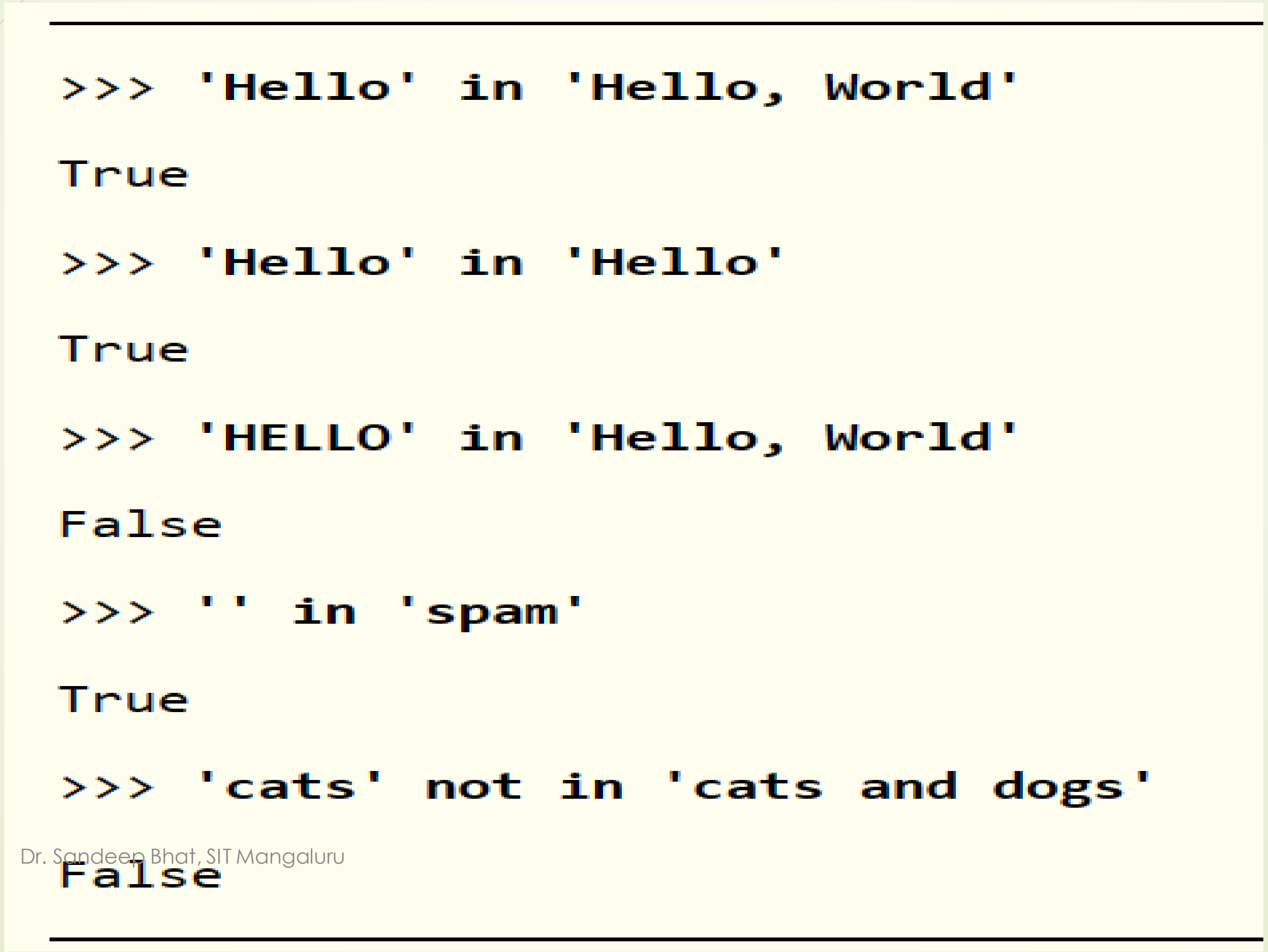
```
>>> spam = 'Hello, world!'
>>> fizz = spam[0:5]
>>> fizz
'Hello'
```

The in and not in Operators with Strings

- **The in and not in operators can be used with strings just like with list values.**

- **An expression with two strings joined using in or not in will evaluate to a Boolean True or False.**

Eg.



```
>>> 'Hello' in 'Hello, World'
True
>>> 'Hello' in 'Hello'
True
>>> 'HELLO' in 'Hello, World'
False
>>> '' in 'spam'
True
>>> 'cats' not in 'cats and dogs'
False
```

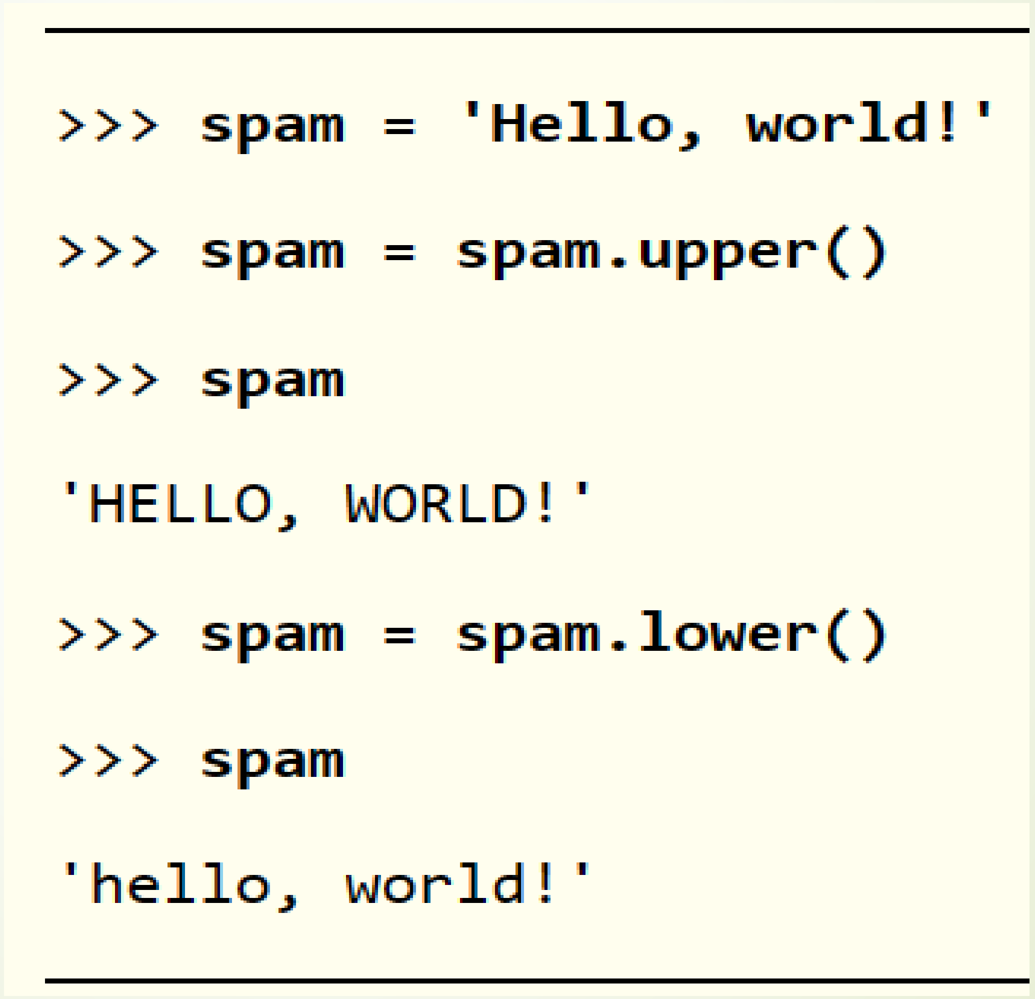
B. USEFUL STRING METHODS

The upper(), lower(), isupper(), and islower() Methods

- The upper() and lower() string methods return a new string where all the letters in the original string have been converted to uppercase or lowercase, respectively.

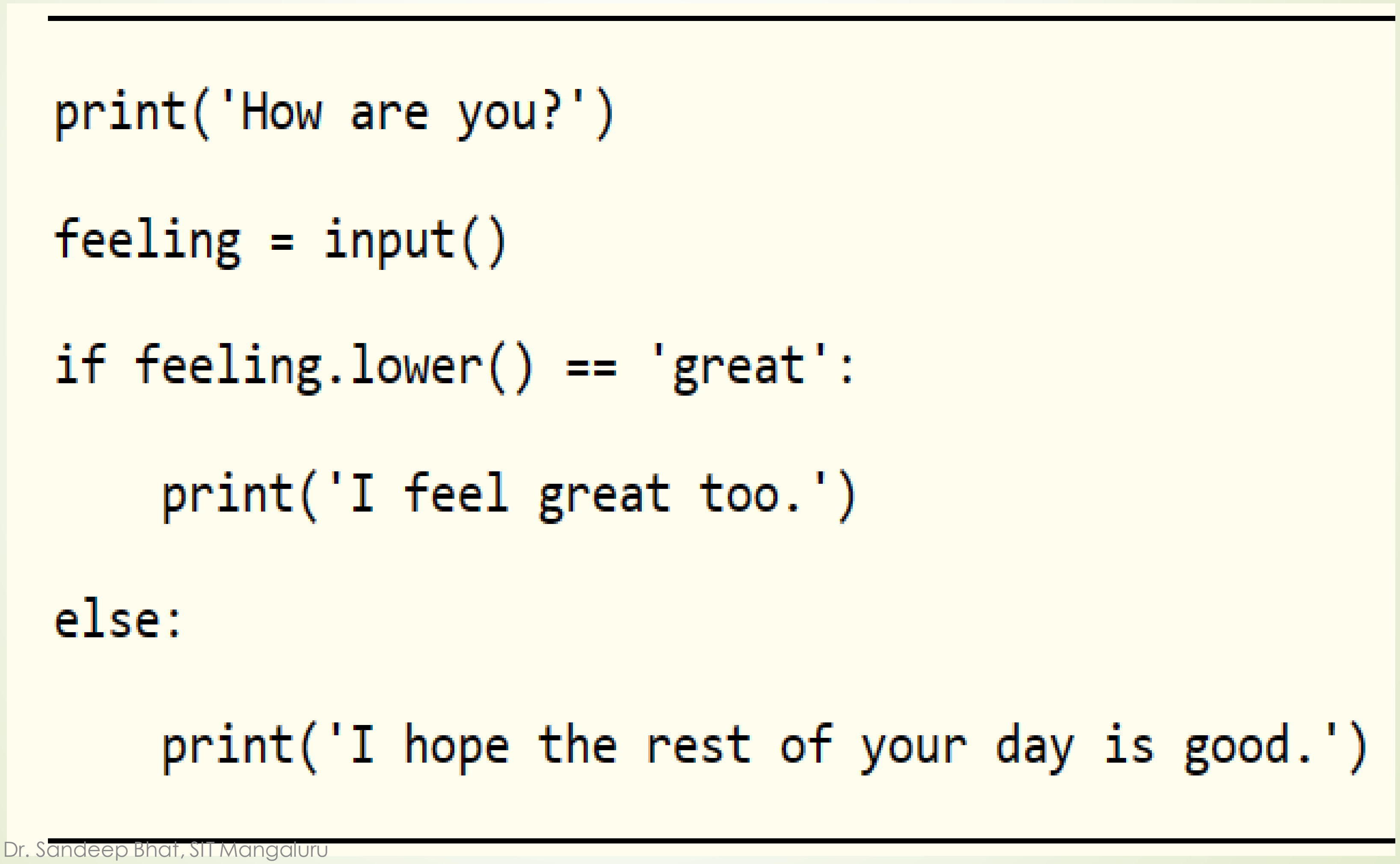
- Nonletter characters in the string remain unchanged.

Eg.



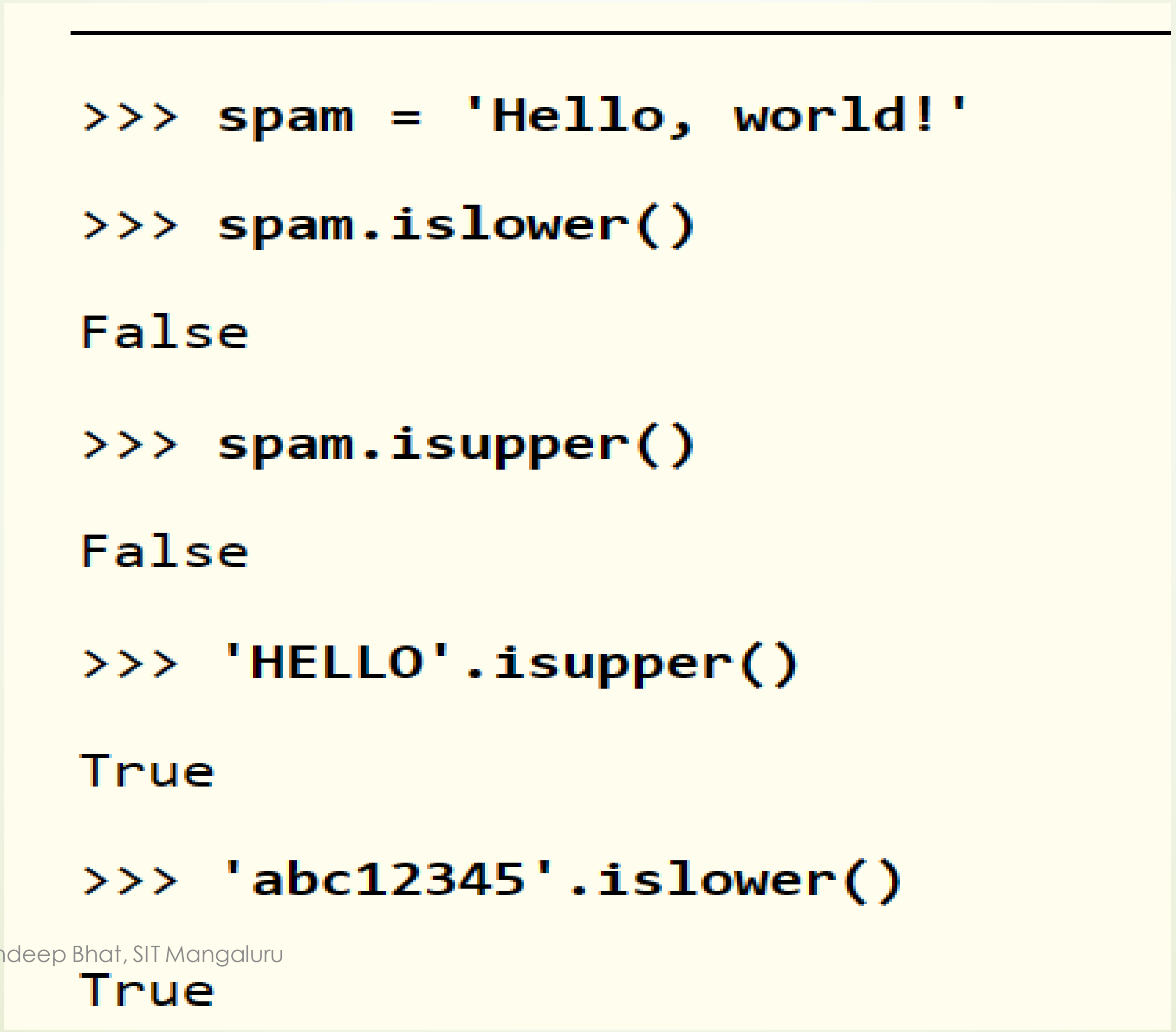
```
>>> spam = 'Hello, world!'
>>> spam = spam.upper()
>>> spam
'HELLO, WORLD!'
>>> spam = spam.lower()
>>> spam
'hello, world!'
```

Eg.2



```
print('How are you?')

feeling = input()

if feeling.lower() == 'great':

    print('I feel great too.')

else:

    print('I hope the rest of your day is good.')
```

Eg.3

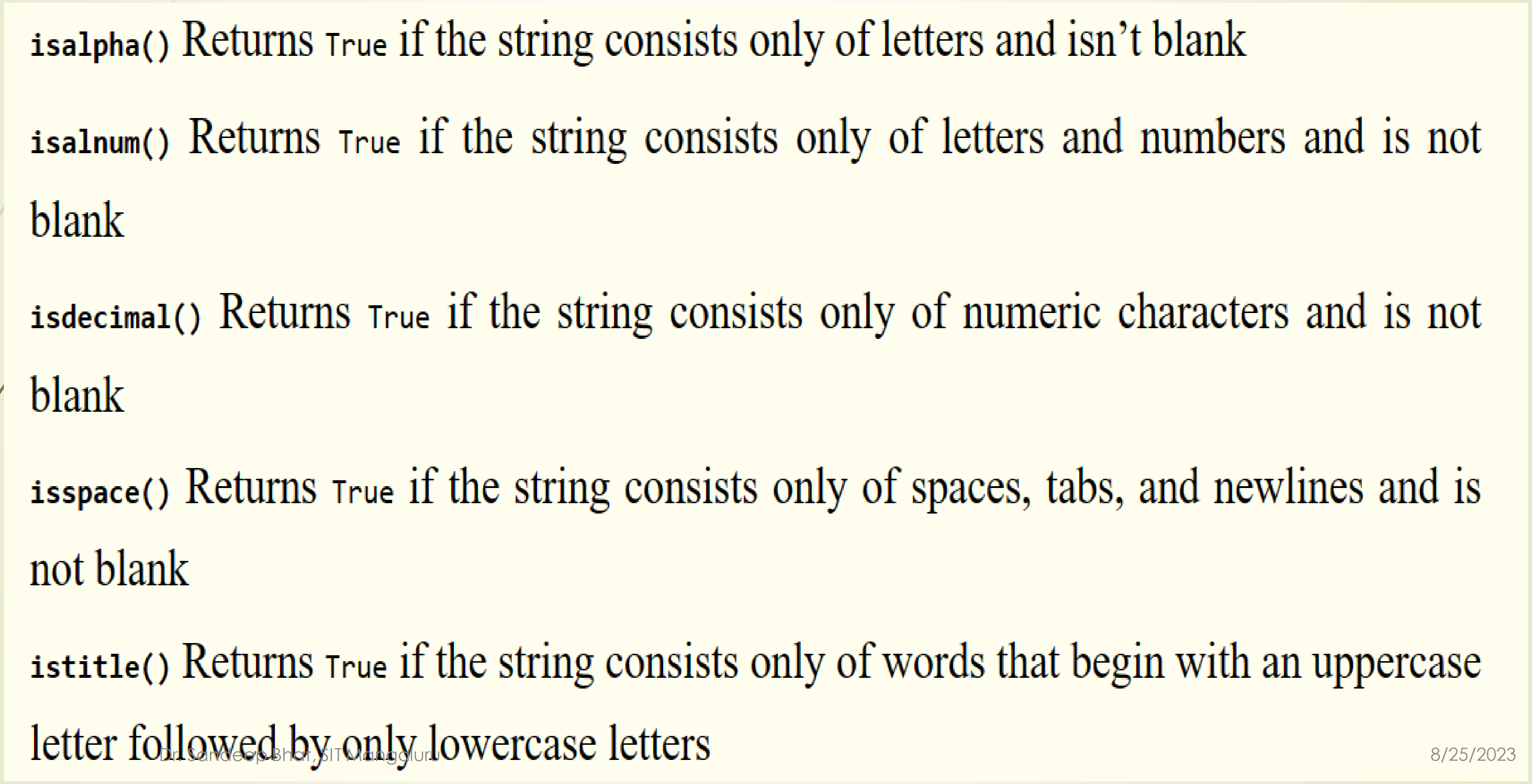


```
>>> spam = 'Hello, world!'
>>> spam.islower()
False
>>> spam.isupper()
False
>>> 'HELLO'.isupper()
True
>>> 'abc12345'.islower()
True
```

The isX() Methods

- Along with islower() and isupper(), there are several other string methods that have names beginning with the word *is*.
- These methods return a Boolean value that describes the nature of the string.
- Here are some common isX string methods:

Cont..



`isalpha()` Returns `True` if the string consists only of letters and isn't blank

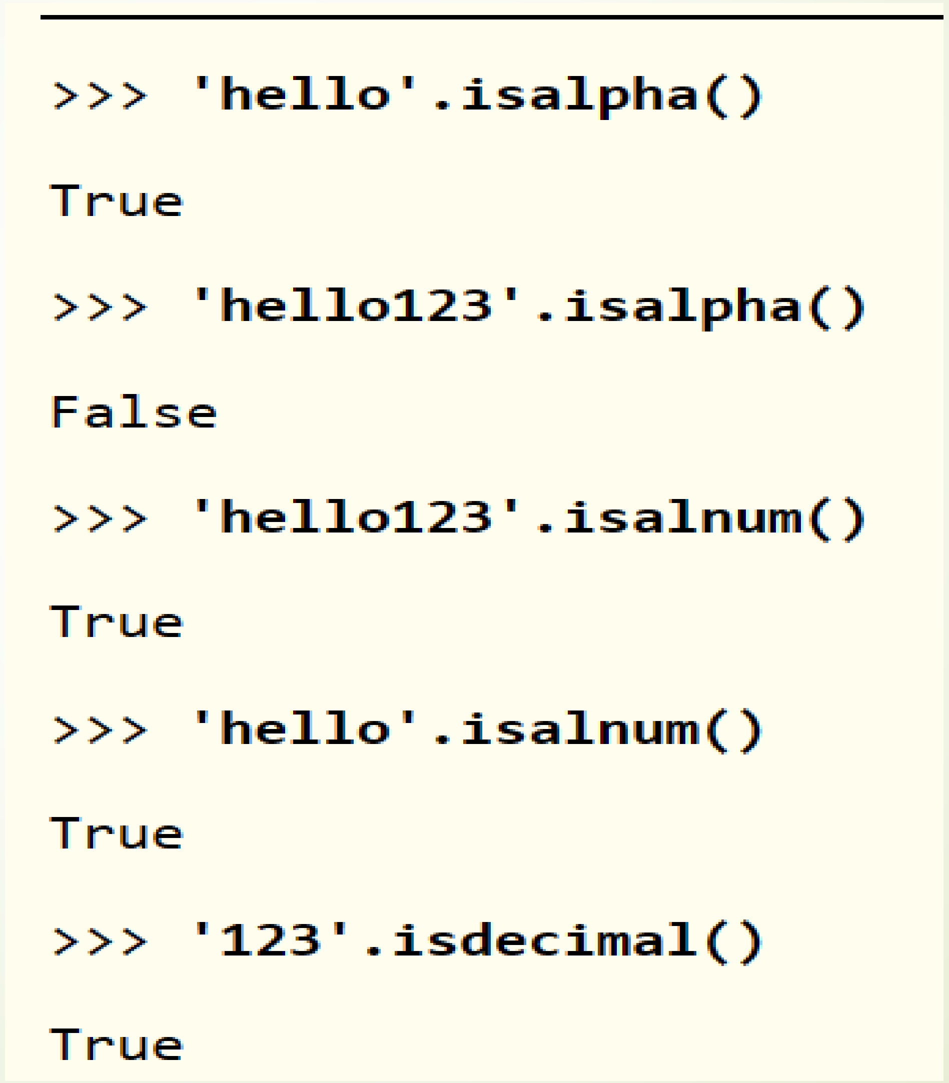
`isalnum()` Returns `True` if the string consists only of letters and numbers and is not blank

`isdecimal()` Returns `True` if the string consists only of numeric characters and is not blank

`isspace()` Returns `True` if the string consists only of spaces, tabs, and newlines and is not blank

`istitle()` Returns `True` if the string consists only of words that begin with an uppercase letter followed by only lowercase letters

Eg.

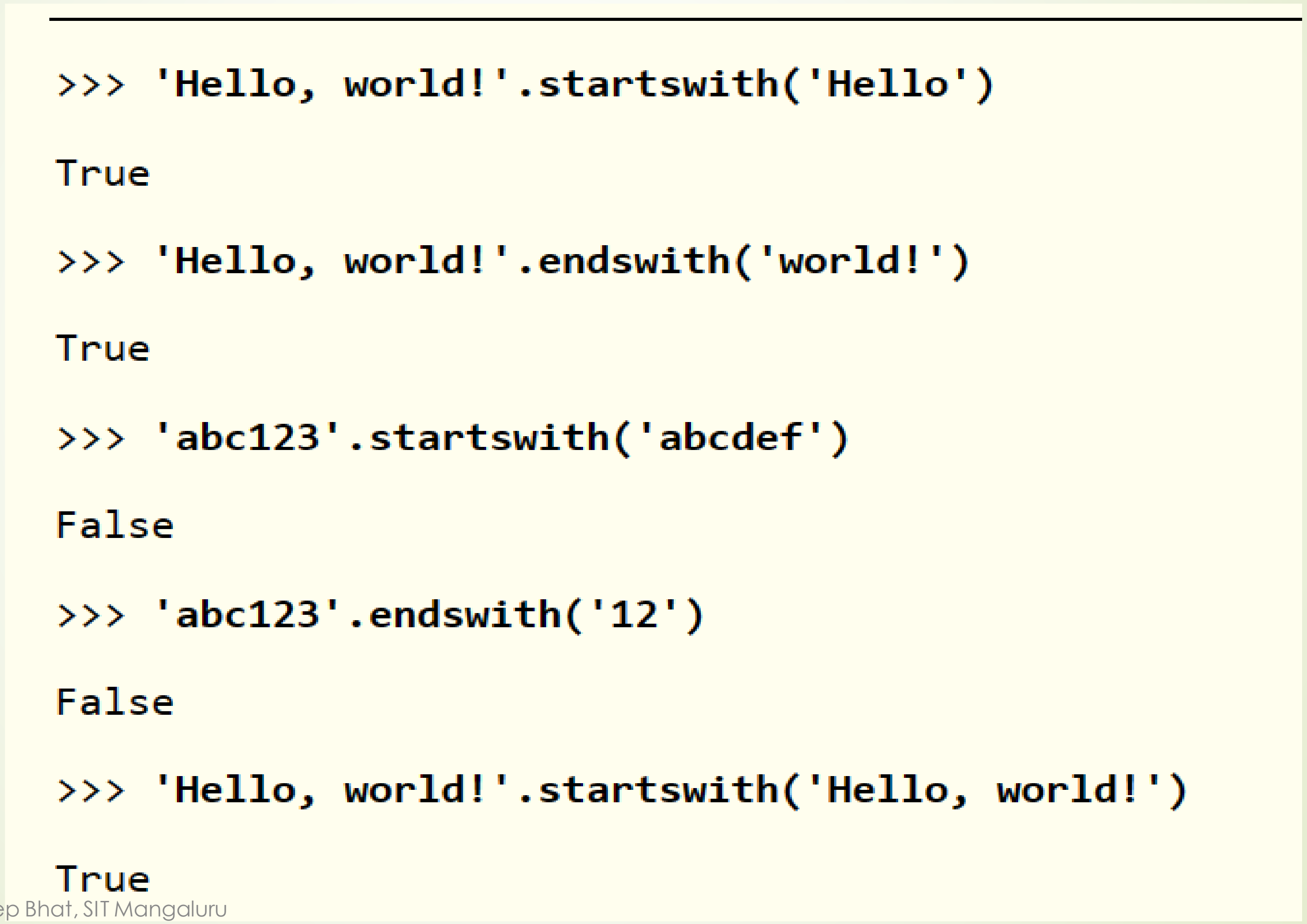


```
>>> 'hello'.isalpha()
True
>>> 'hello123'.isalpha()
False
>>> 'hello123'.isalnum()
True
>>> 'hello'.isalnum()
True
>>> '123'.isdecimal()
True
```

The startswith() and endswith() Methods

- **The startswith() and endswith() methods return True if the string value they are called on begins or ends (respectively) with the string passed to the method; otherwise, they return False.**

Cont..

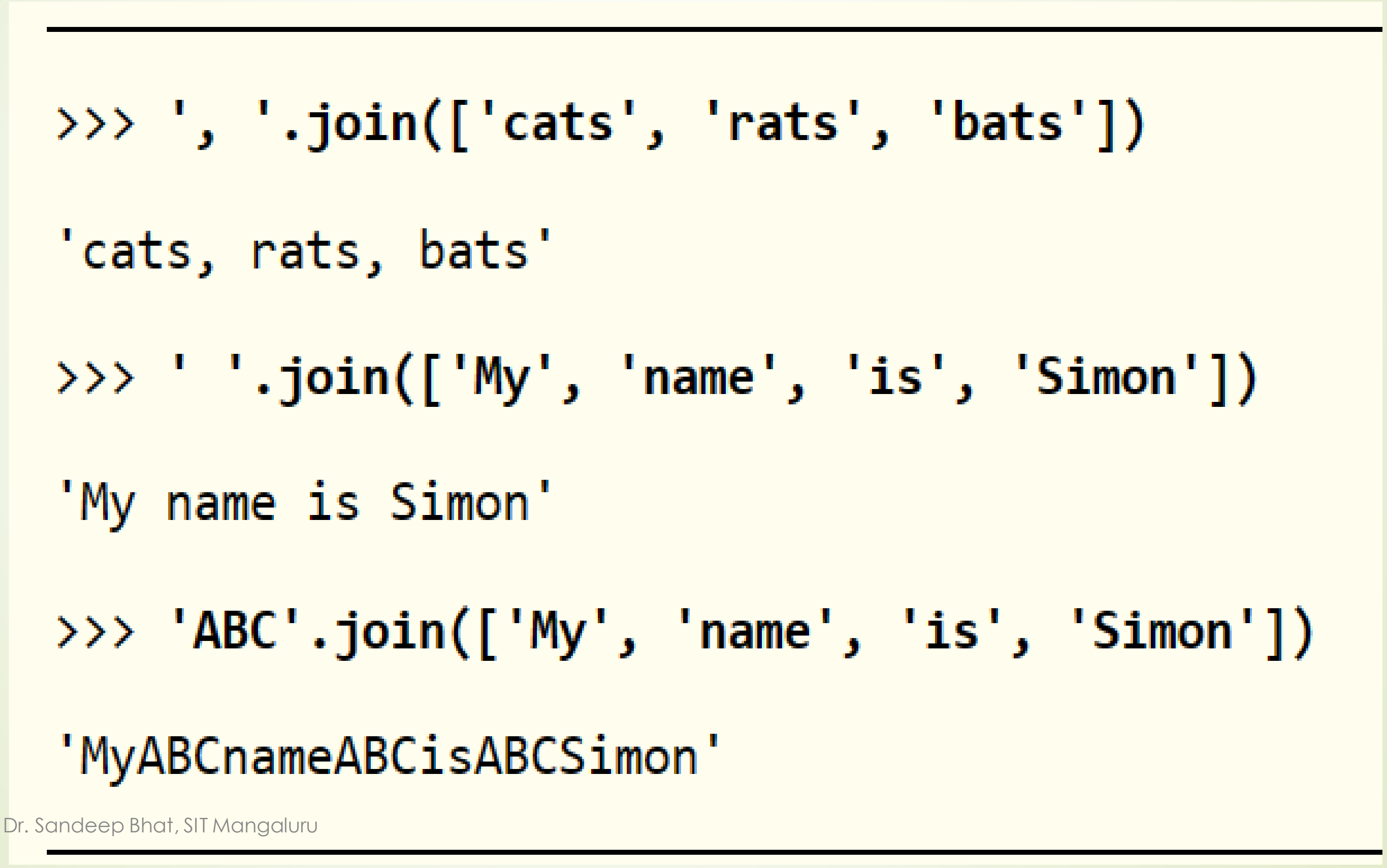


```
>>> 'Hello, world!'.startswith('Hello')
True
>>> 'Hello, world!'.endswith('world!')
True
>>> 'abc123'.startswith('abcdef')
False
>>> 'abc123'.endswith('12')
False
>>> 'Hello, world!'.startswith('Hello, world!')
True
```

The join() and split() Methods

- **The join() method is useful when you have a list of strings that need to be joined together into a single string value.**
- **The join() method is called on a string, gets passed a list of strings, and returns a string.**
- **The returned string is the concatenation of each string in the passed-in list.**

Eg.

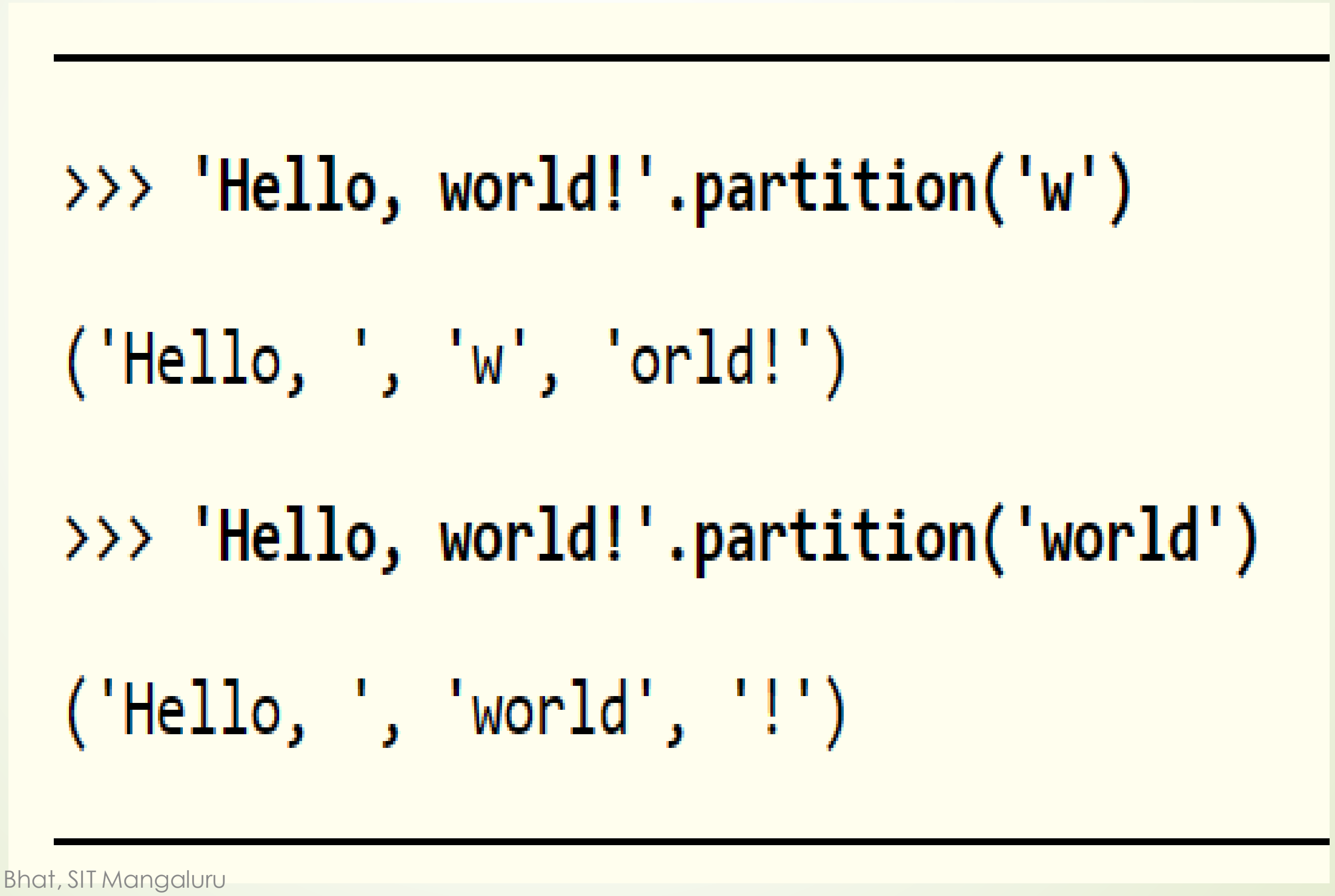


```
>>> ', '.join(['cats', 'rats', 'bats'])
'cats, rats, bats'
>>> ' '.join(['My', 'name', 'is', 'Simon'])
'My name is Simon'
>>> 'ABC'.join(['My', 'name', 'is', 'Simon'])
'MyABCnameABCisABCSimon'
```

Splitting Strings with the partition() Method

- **The partition() string method can split a string into the text before and after a separator string.**
- **This method searches the string it is called on for the separator string it is passed, and returns a tuple of three substrings for the “before,” “separator,” and “after” substrings.**

Eg.

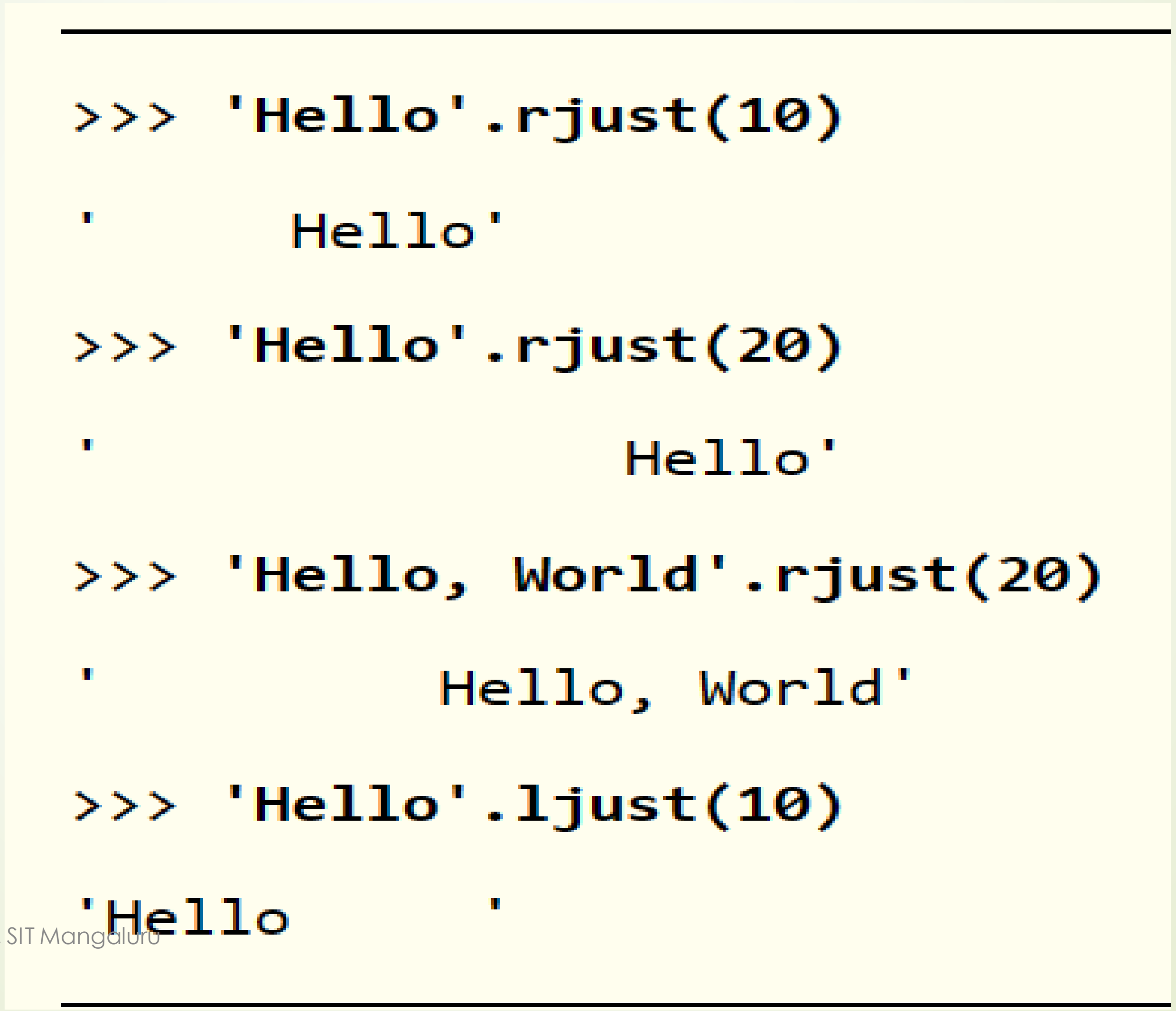


```
>>> 'Hello, world!'.partition('w')
('Hello, ', 'w', 'orld!')
>>> 'Hello, world!'.partition('world')
('Hello, ', 'world', '!')
```

Justifying Text with the rjust(), ljust(), and center() Methods

- The rjust() and ljust() string methods return a padded version of the string they are called on, with spaces inserted to justify the text.
- The first argument to both methods is an integer length for the justified string.

Eg.



```
>>> 'Hello'.rjust(10)
'     Hello'
>>> 'Hello'.rjust(20)
'               Hello'
>>> 'Hello, World'.rjust(20)
'        Hello, World'
>>> 'Hello'.ljust(10)
'Hello     '
```

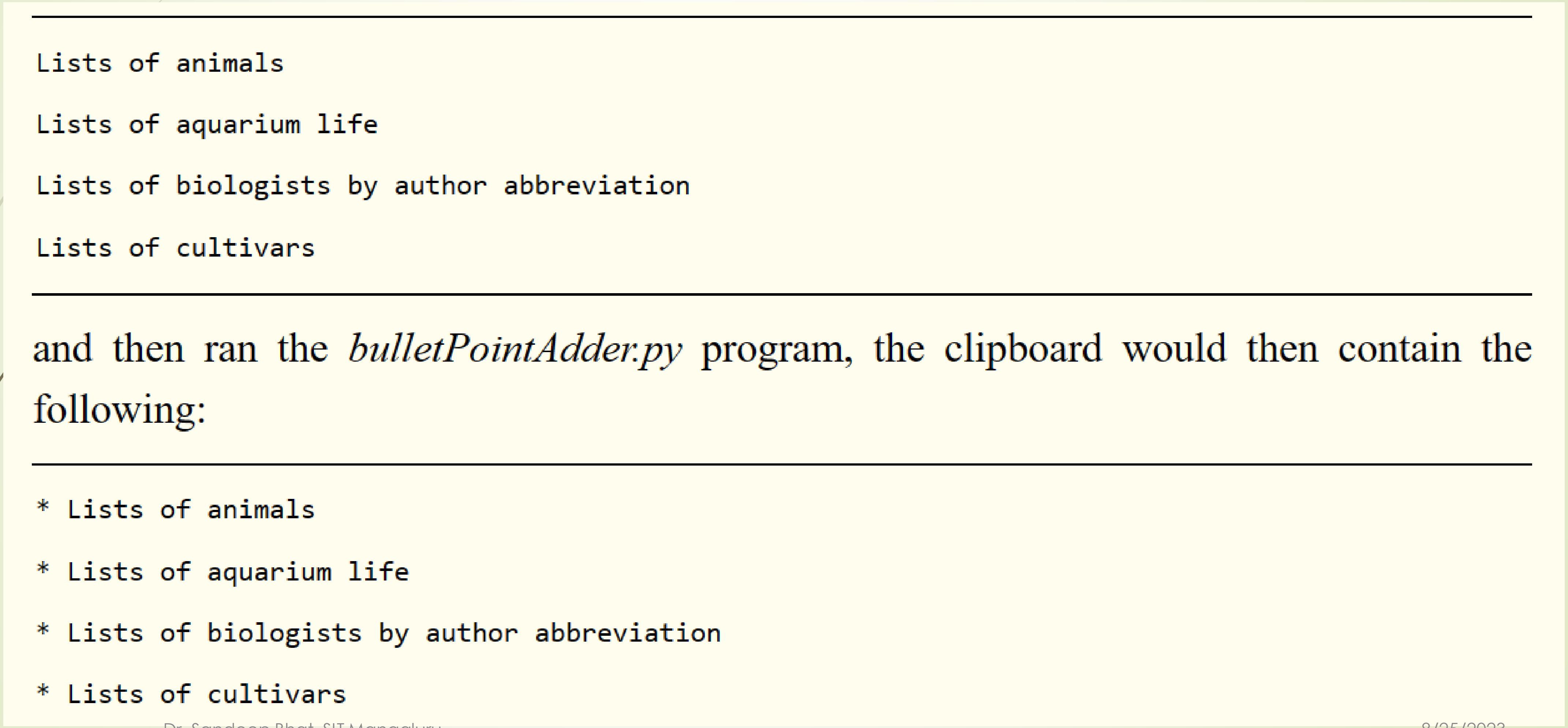
Cont..

- 'Hello'.rjust(10) says that we want to right-justify 'Hello' in a string of total length 10.
- 'Hello' is five characters, so five spaces will be added to its left, giving us a string of 10 characters with 'Hello' justified right.

Project: Adding Bullets to Wiki Markup

- When editing a Wikipedia article, you can create a bulleted list by putting each list item on its own line and placing a star in front.
- But say you have a really large list that you want to add bullet points to. You could just type those stars at the beginning of each line, one by one.
- Or you could automate this task with a short Python script.

Eg.

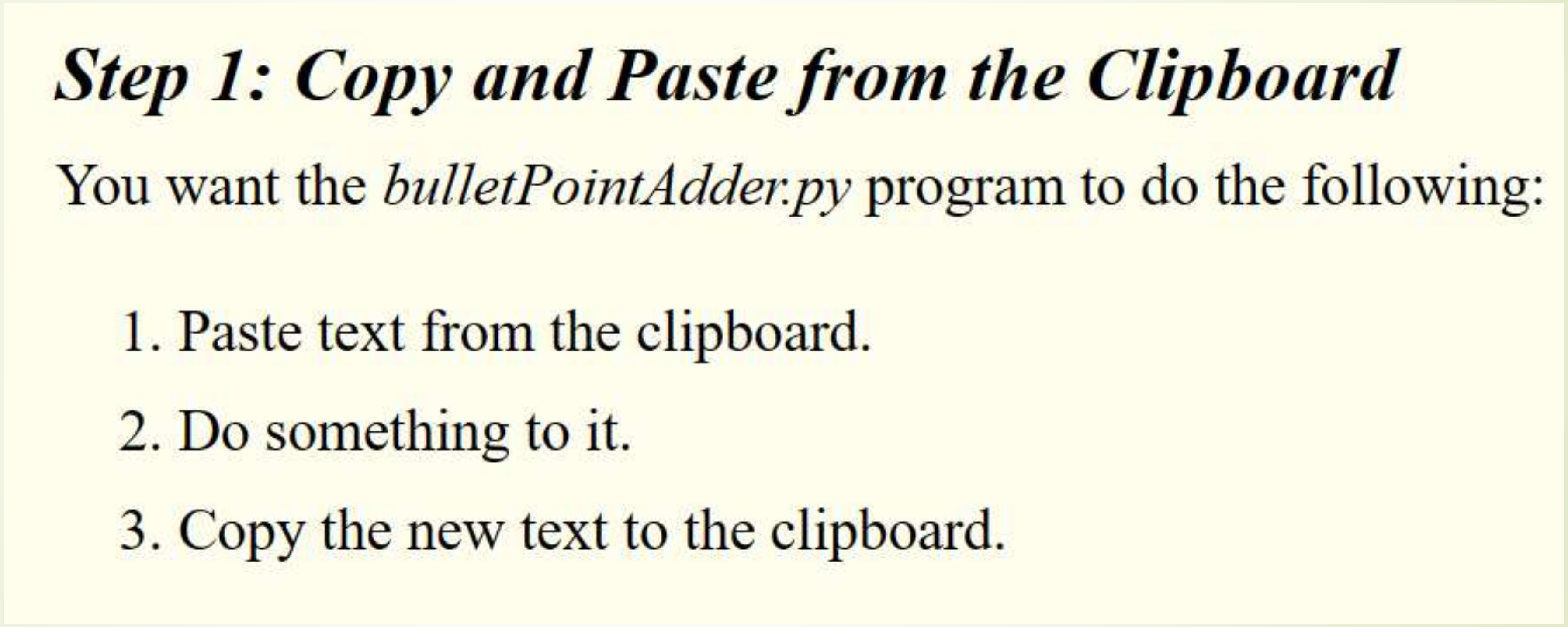


```
Lists of animals
Lists of aquarium life
Lists of biologists by author abbreviation
Lists of cultivars
```

and then ran the *bulletPointAdder.py* program, the clipboard would then contain the following:

```
* Lists of animals
* Lists of aquarium life
* Lists of biologists by author abbreviation
* Lists of cultivars
```

Cont..



Step 1: Copy and Paste from the Clipboard

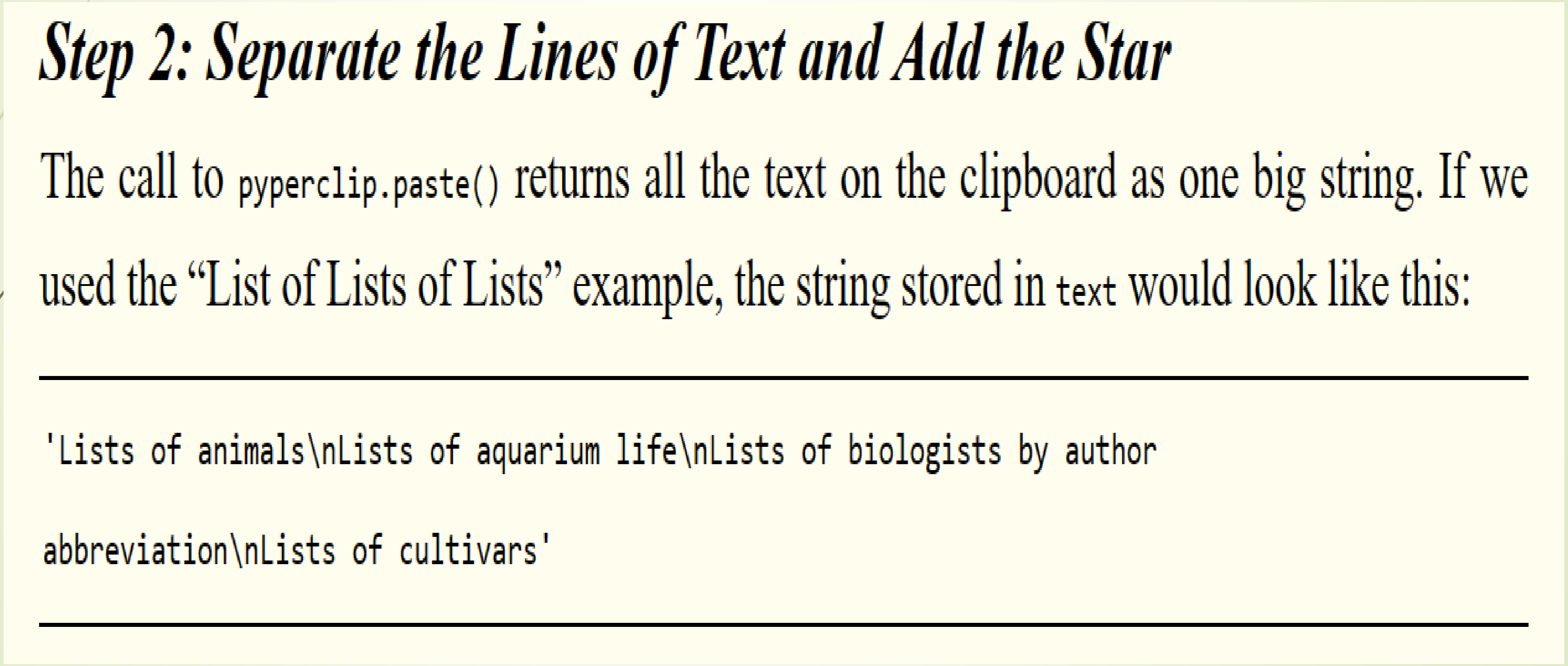
You want the *bulletPointAdder.py* program to do the following:

1. Paste text from the clipboard.
2. Do something to it.
3. Copy the new text to the clipboard.

Cont..

```
#! python3

# bulletPointAdder.py - Adds Wikipedia bullet points to the start

# of each line of text on the clipboard.


import pyperclip

text = pyperclip.paste()


# TODO: Separate lines and add stars.


pyperclip.copy(text)
```

Step 2: Separate the Lines of Text and Add the Star



Step 2: Separate the Lines of Text and Add the Star

The call to `pyperclip.paste()` returns all the text on the clipboard as one big string. If we used the "List of Lists of Lists" example, the string stored in `text` would look like this:

```
'Lists of animals\nLists of aquarium life\nLists of biologists by author
abbreviation\nLists of cultivars'
```

Cont..

- The \n newline characters in this string cause it to be displayed with multiple lines when it is printed or pasted from the clipboard.
- There are many “lines” in this one string value.
- You want to add a star to the start of each of these lines.

Cont..

```
import pyperclip
text = pyperclip.paste()


# Separate lines and add stars.
lines = text.split('\n')
for i in range(len(lines)):    # loop through all indexes in the "lines" list
    lines[i] = '* ' + lines[i] # add star to each string in "lines" list


pyperclip.copy(text)
```

Step 3: Join the Modified Lines

- The lines list now contains modified lines that start with stars.
- But pyperclip.copy() is expecting a single string value, however, not a list of string values.
- To make this single string value, pass lines into the join() method to get a single string joined from the list's strings.

Cont..

```
import pyperclip

text = pyperclip.paste()



# Separate lines and add stars.

lines = text.split('\n')

for i in range(len(lines)):    # loop through all indexes for "lines" list

    lines[i] = '* ' + lines[i] # add star to each string in "lines" list

text = '\n'.join(lines)

pyperclip.copy(text)
```

Cont..

- **When this program is run, it replaces the text on the clipboard with text that has stars at the start of each line.**

- **Now the program is complete, and you can try running it with text copied to the clipboard.**

Project: Multi-Clipboard Automatic Messages

- If you’ve responded to a large number of emails with similar phrasing, you’ve probably had to do a lot of repetitive typing.
- Maybe you keep a text document with these phrases so you can easily copy and paste them using the clipboard.
- But your clipboard can only store one message at a time, which isn’t very convenient.
- Let’s make this process a bit easier with a program that stores multiple phrases.

➢ Step 1: Program Design and Data Structures

- You want to be able to run this program with a command line argument that is a short key phrase—for instance, *agree* or *busy*.

- The message associated with that key phrase will be copied to the clipboard so that the user can paste it into an email.

- This way, the user can have long, detailed messages without having to retype them.

The code:

```
#! python3

# mclip.py - A multi-clipboard program.


TEXT = {'agree': """Yes, I agree. That sounds fine to me.""",

        'busy': """Sorry, can we do this later this week or next week?""",

        'upsell': """Would you consider making this a monthly donation?"""}
```

Step 2: Handle Command Line Arguments

- The command line arguments will be stored in the variable sys.argv.
- The first item in the sys.argv list should always be a string containing the program's filename ('mclip.py'), and the second item should be the first command line argument.
- For this program, this argument is the key phrase of the message you want.

The Code:

```
#! python3
# mclip.py - A multi-clipboard program.

TEXT = {'agree': """Yes, I agree. That sounds fine to me.""",
        'busy': """Sorry, can we do this later this week or next week?""",
        'upsell': """Would you consider making this a monthly donation?"""}

import sys
if len(sys.argv) < 2:
    print('Usage: python mclip.py [keyphrase] - copy phrase text')
    sys.exit()

keyphrase = sys.argv[1]    # first command line arg is the keyphrase
```

Step 3: Copy the Right Phrase

- Now that the key phrase is stored as a string in the variable keyphrase, you need to see whether it exists in the TEXT dictionary as a key.

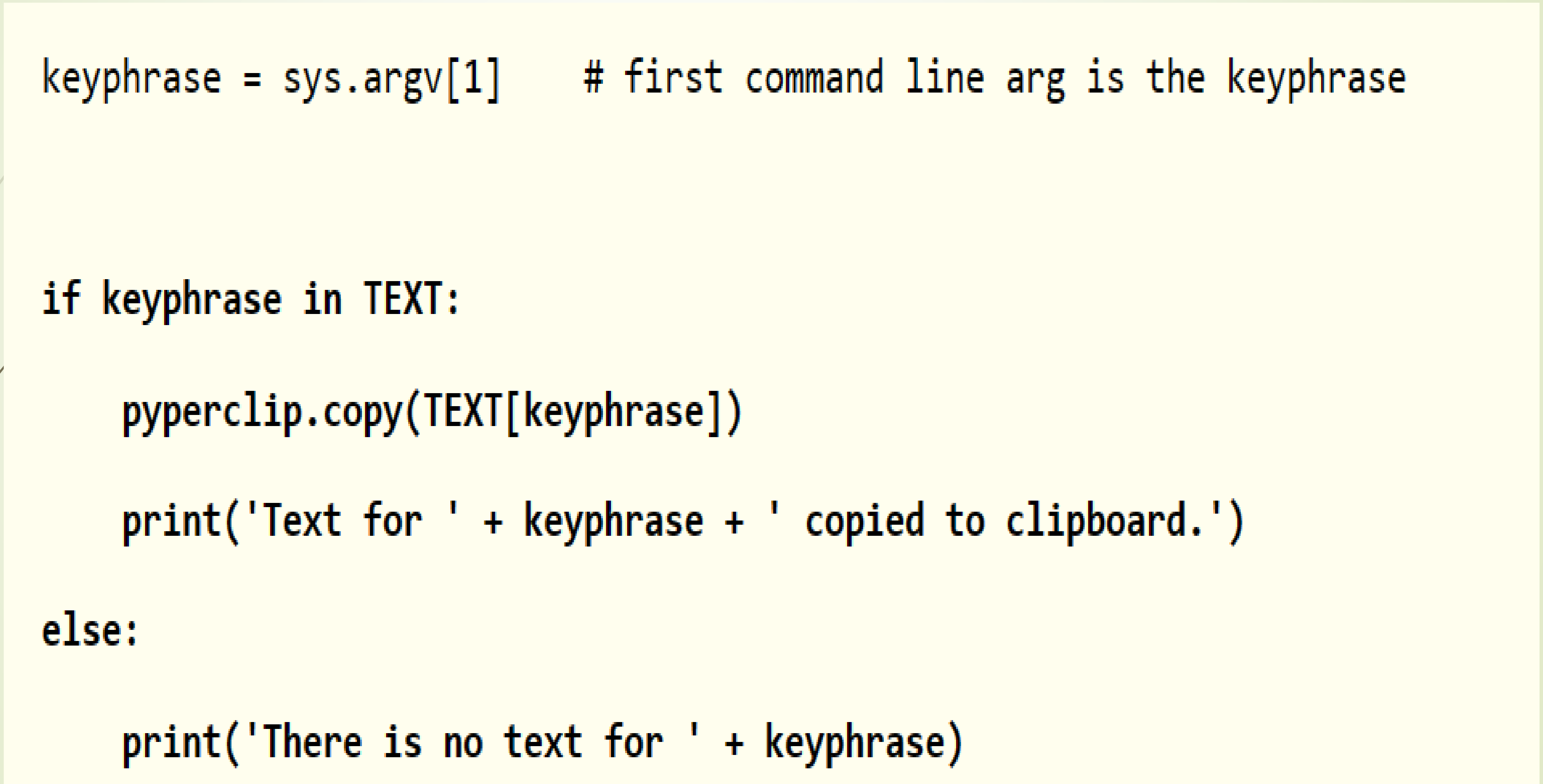
- If so, you want to copy the key's value to the clipboard using pyperclip.copy(). (Since you're using the pyperclip module, you need to import it.)

- Note that you don't actually *need* the keyphrase variable; you could just use sys.argv[1] everywhere keyphrase is used in this program.

Code:

```
#! python3

# mclip.py - A multi-clipboard program.


TEXT = {'agree': """Yes, I agree. That sounds fine to me.""",

        'busy': """Sorry, can we do this later this week or next week?""",

        'upsell': """Would you consider making this a monthly donation?"""}


import sys, pyperclip

if len(sys.argv) < 2:

    print('Usage: py mclip.py [keyphrase] - copy phrase text')

    sys.exit()
```

Cont..



```
keyphrase = sys.argv[1]    # first command line arg is the keyphrase


if keyphrase in TEXT:
    pyperclip.copy(TEXT[keyphrase])
    print('Text for ' + keyphrase + ' copied to clipboard.')
else:
    print('There is no text for ' + keyphrase)
```

Cont..

- This new code looks in the TEXT dictionary for the key phrase.
- If the key phrase is a key in the dictionary, we get the value corresponding to that key, copy it to the clipboard, and print a message saying that we copied the value.
- Otherwise, we print a message saying there’s no key phrase with that name.

Reading and Writing Files

Files and File Paths

- A file has two key properties: a *filename* (usually written as one word) and a *path*. The path specifies the location of a file on the computer.

- For example, there is a file on my Windows laptop with the filename *project.docx* in the path *C:\Users\Al\Documents*.

- The part of the filename after the last period is called the file's *extension* and tells you a file's type.

Cont..

- **The filename *project.docx* is a Word document, and *Users*, *Al*, and *Documents* all refer to *folders* (also called *directories*).**
- **Folders can contain files and other folders.**
- **For example, *project.docx* is in the *Documents* folder, which is inside the *Al* folder, which is inside the *Users* folder.**
- **Figure 9-1 shows this folder organization.**

Cont..

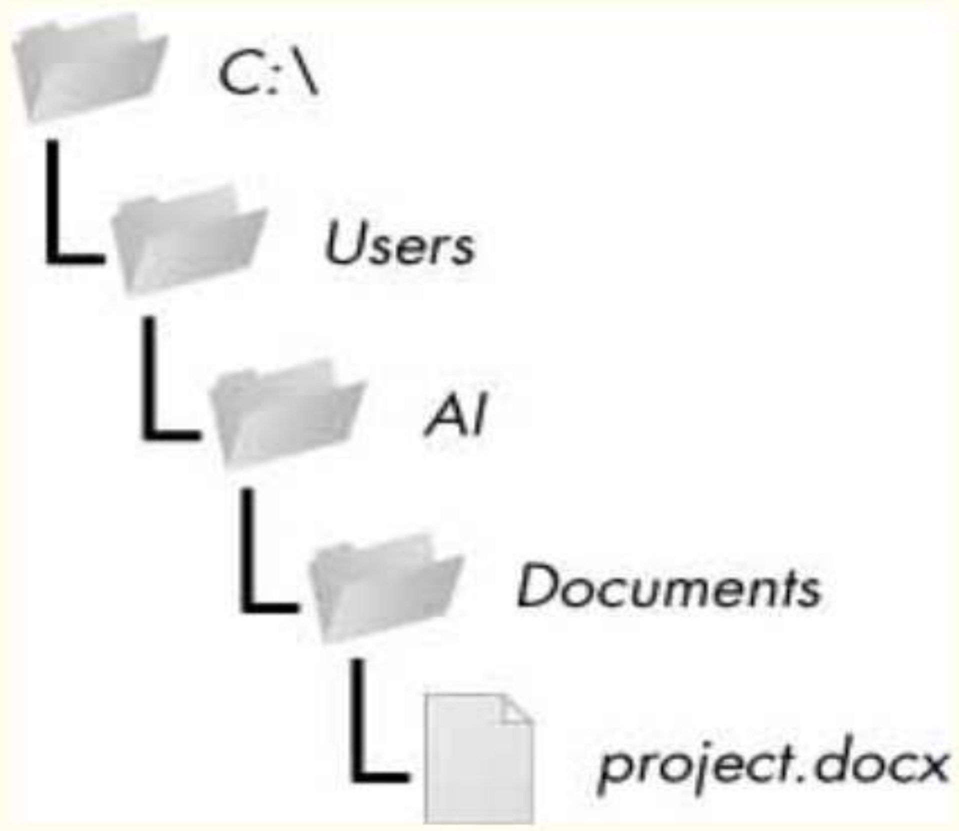


Figure 9-1: A file in a hierarchy of folders

Cont..

- The C:\ part of the path is the *root folder*, which contains all other folders.
- On Windows, the root folder is named C:\ and is also called the C: *drive*.
- On macOS and Linux, the root folder is /.

Cont..

- **Additional *volumes*, such as a DVD drive or USB flash drive, will appear differently on different operating systems.**

- **On Windows, they appear as new, lettered root drives, such as *D:*\ or *E:*\.**

- **On macOS, they appear as new folders under the /*Volumes* folder.**

- **On Linux, they appear as new folders under the /*mnt* (“mount”) folder.**

Backslash on Windows and Forward Slash on macOS and Linux

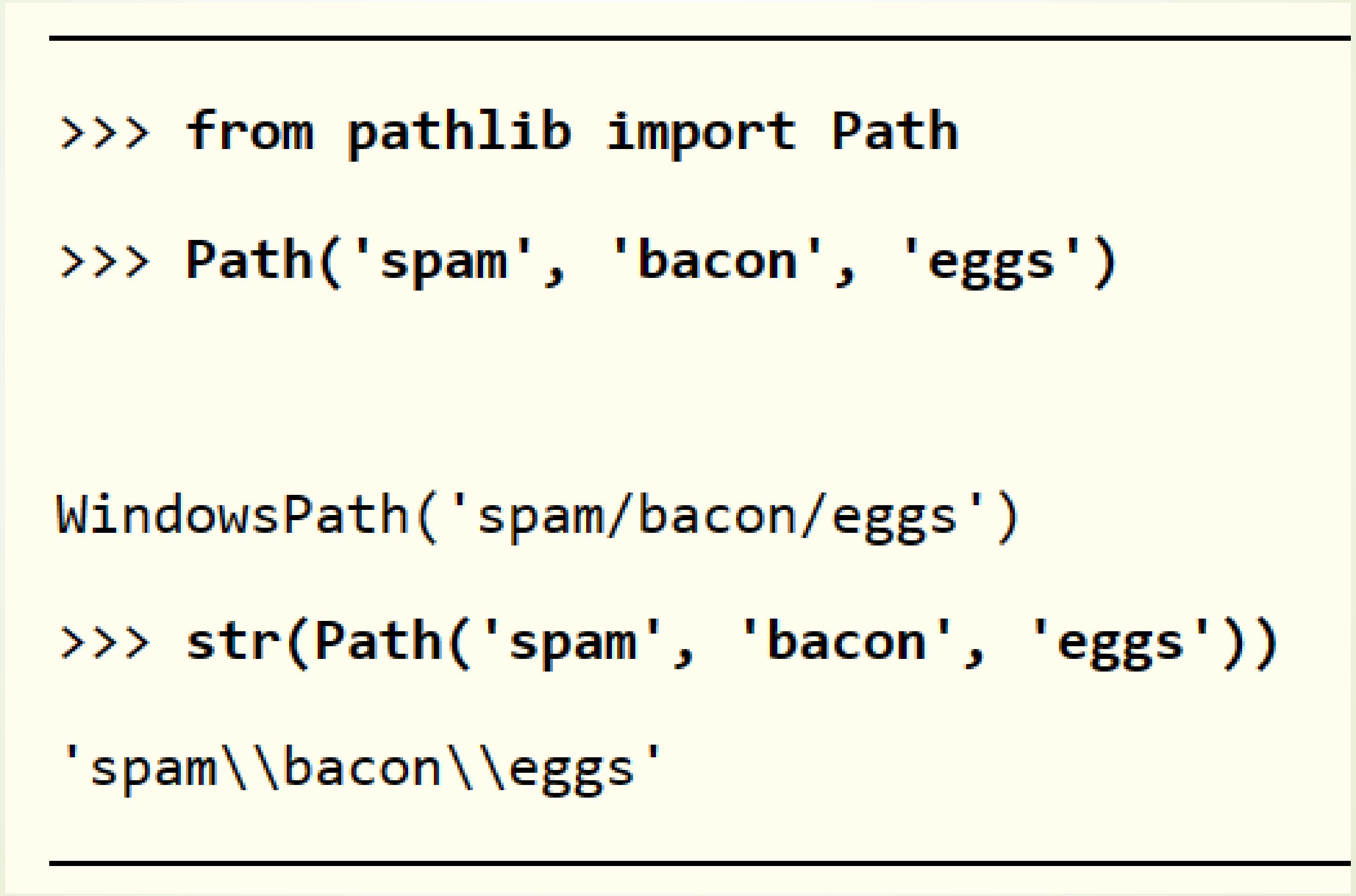
- On Windows, paths are written using backslashes (\) as the separator between folder names.
- The macOS and Linux operating systems, however, use the forward slash (/) as their path separator.
- If you want your programs to work on all operating systems, you will have to write your Python scripts to handle both cases.

Cont..

- Fortunately, this is simple to do with the **Path()** function in the **pathlib** module.

- If you pass it the string values of individual file and folder names in your path, Path() will return a string with a file path using the correct path separators.

Eg.



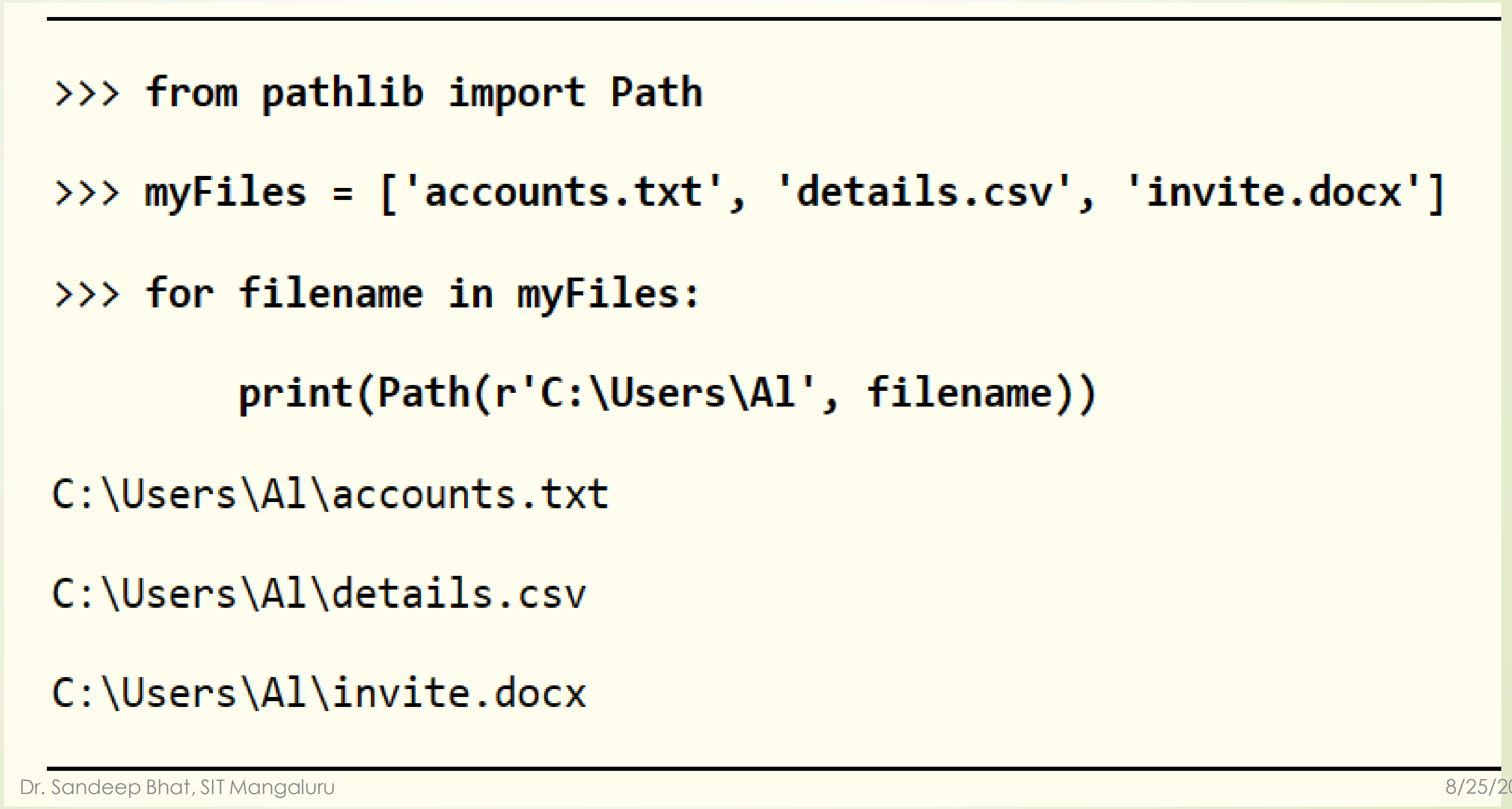
```
>>> from pathlib import Path
>>> Path('spam', 'bacon', 'eggs')


WindowsPath('spam/bacon/eggs')
>>> str(Path('spam', 'bacon', 'eggs'))
'spam\\bacon\\eggs'
```

Cont..

- Note that the convention for importing pathlib is to run from pathlib import Path, since otherwise we'd have to enter pathlib.

- Path everywhere Path shows up in our code. Not only is this extra typing redundant, but it's also redundant.

- **For example, the following code joins names from a list of filenames to the end of a folder's name:**



```
>>> from pathlib import Path
>>> myFiles = ['accounts.txt', 'details.csv', 'invite.docx']
>>> for filename in myFiles:
        print(Path(r'C:\Users\Al', filename))
C:\Users\Al\accounts.txt
C:\Users\Al\details.csv
C:\Users\Al\invite.docx
```

Cont..

- **On Windows, the backslash separates directories, so you can't use it in filenames.**
- **However, you can use backslashes in filenames on macOS and Linux.**
- **So while Path(r'spam\eggs') refers to two separate folders (or a file *eggs* in a folder *spam*) on Windows, the same command would refer to a single folder (or file) named *spam\eggs* on macOS and Linux.**
- **For this reason, it's usually a good idea to always use forward slashes in your Python code.**

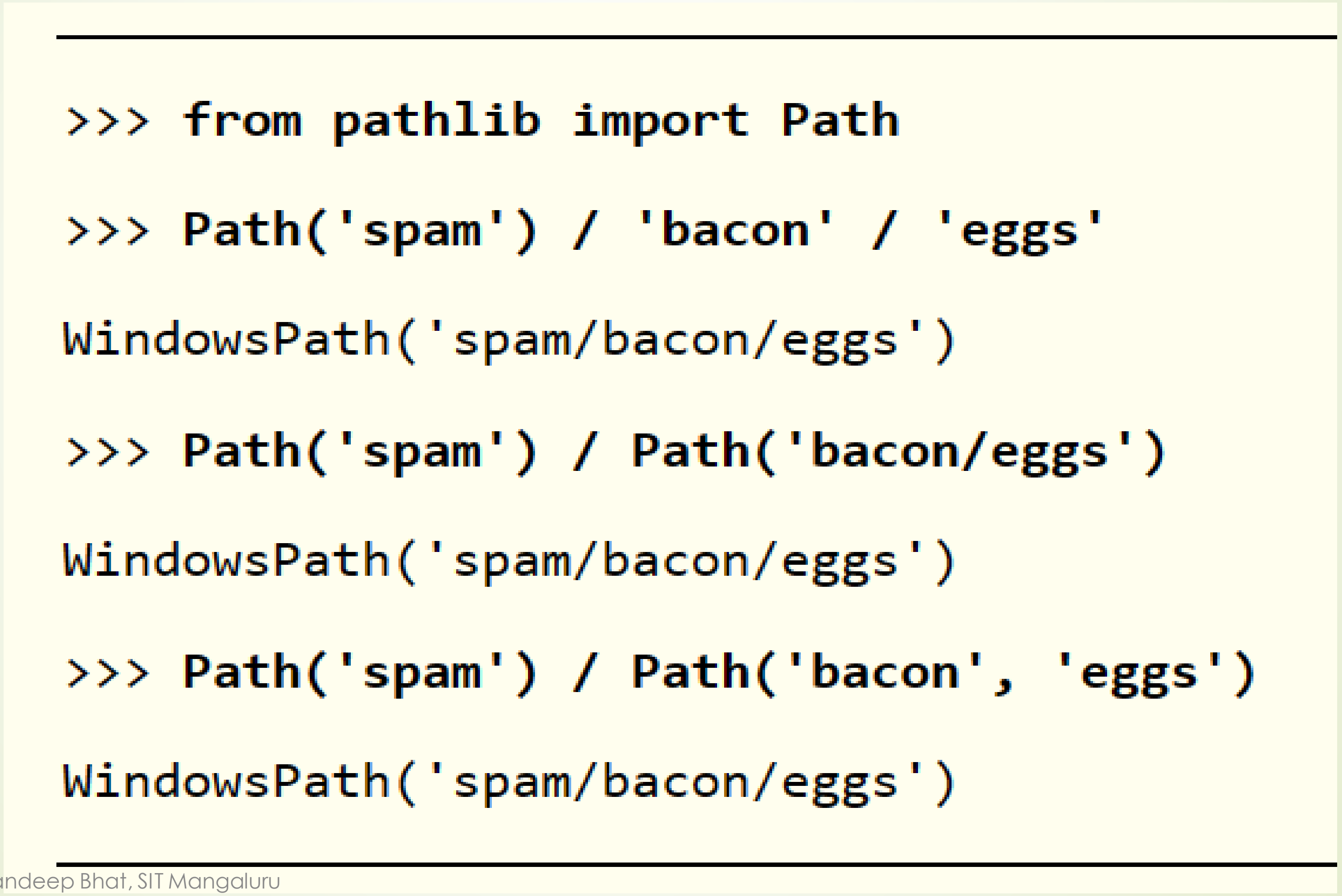
Using the / Operator to Join Paths

- We normally use the + operator to add two integer or floating-point numbers, such as in the expression 2 + 2, which evaluates to the integer value 4.

- But we can also use the + operator to concatenate two string values, like the expression 'Hello' + 'World', which evaluates to the string value 'HelloWorld'.

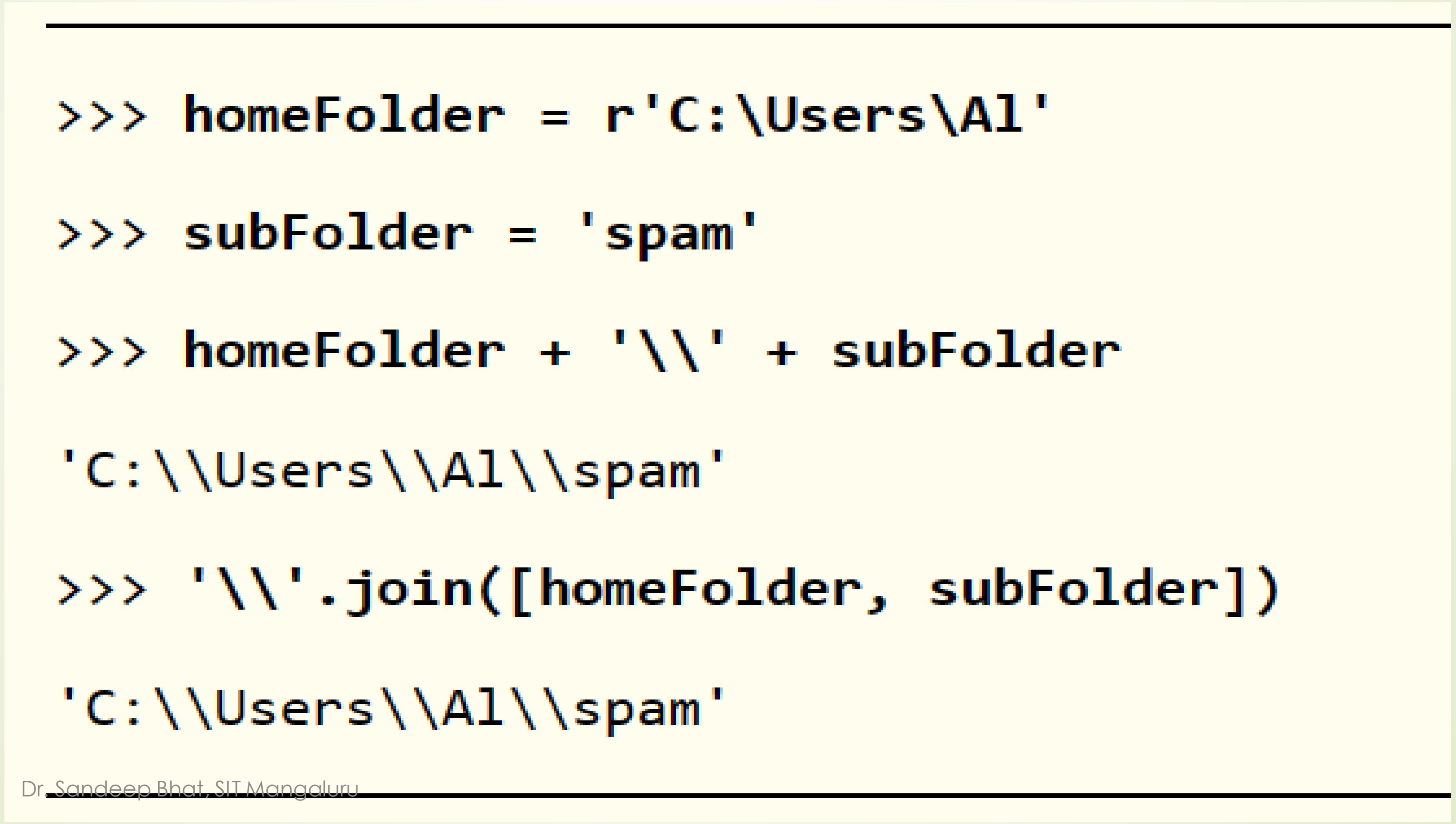
- Similarly, the / operator that we normally use for division can also combine Path objects and strings.

Cont..



```
>>> from pathlib import Path
>>> Path('spam') / 'bacon' / 'eggs'
WindowsPath('spam/bacon/eggs')
>>> Path('spam') / Path('bacon/eggs')
WindowsPath('spam/bacon/eggs')
>>> Path('spam') / Path('bacon', 'eggs')
WindowsPath('spam/bacon/eggs')
```

- **It’s also safer than using string concatenation or the join() method, like we do in this example:**



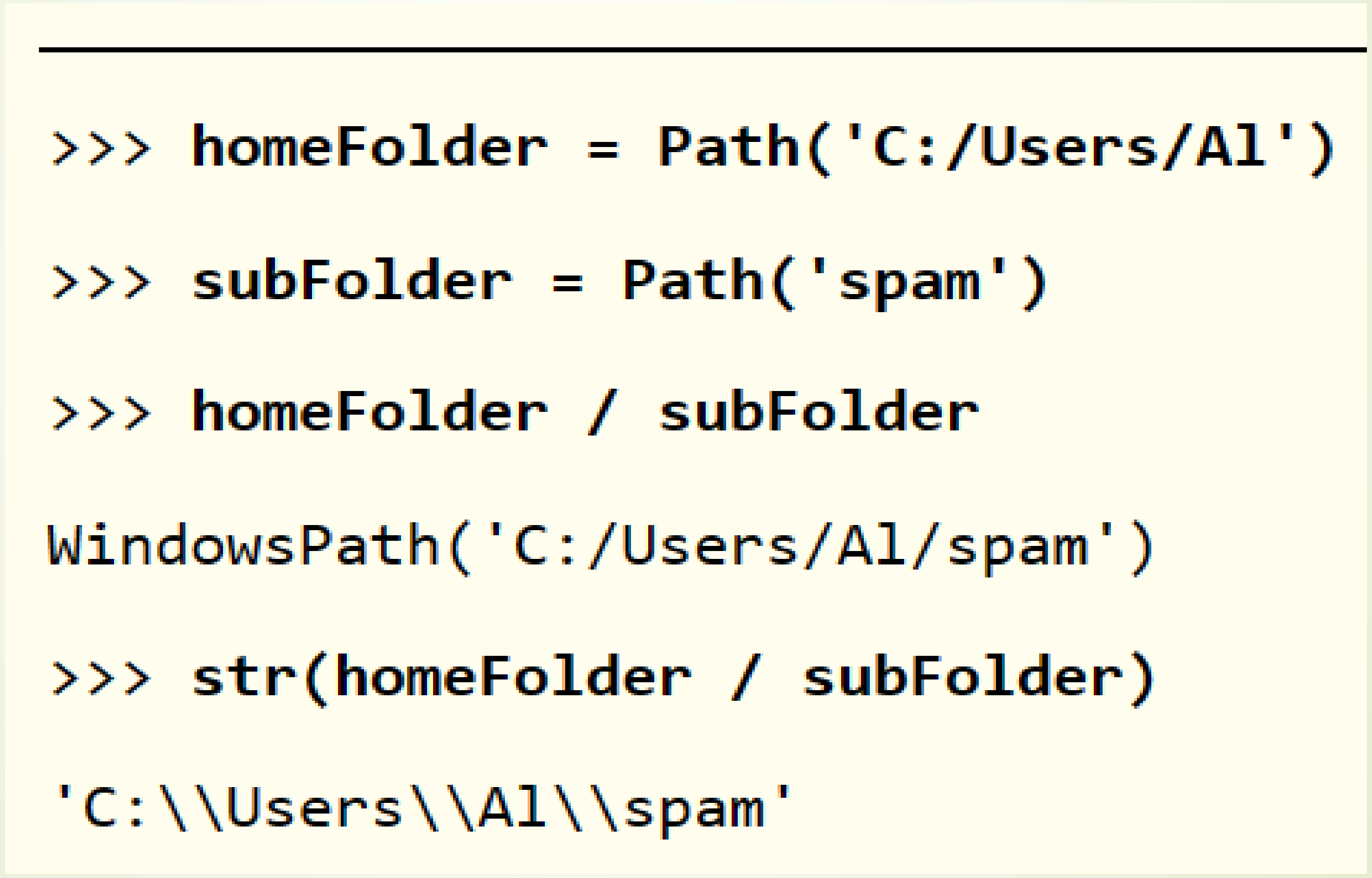
```
>>> homeFolder = r'C:\Users\Al'
>>> subFolder = 'spam'
>>> homeFolder + '\\' + subFolder
'C:\\Users\\Al\\spam'
>>> '\\'.join([homeFolder, subFolder])
'C:\\Users\\Al\\spam'
```

Cont..

- A script that uses this code isn't safe, because its backslashes would only work on Windows.

- You could add an if statement that checks **sys.platform** (which contains a string describing the computer's operating system) to decide what kind of slash to use, but applying this custom code everywhere it's needed can be inconsistent and bug-prone.

- The following example uses this strategy to join the same paths as in the previous example:



```
>>> homeFolder = Path('C:/Users/Al')
>>> subFolder = Path('spam')
>>> homeFolder / subFolder
WindowsPath('C:/Users/Al/spam')
>>> str(homeFolder / subFolder)
'C:\\Users\\Al\\spam'
```

Cont..

- **The only thing you need to keep in mind when using the / operator for joining paths is that one of the first two values must be a Path object.**

os.path module

- In Python, **os.path** is a module that provides functions for working with file paths and directories.

- It allows you to manipulate paths in a platform-independent way, meaning your code will work across different operating systems (Windows, macOS, Linux, etc.) without worrying about the specific path conventions on each platform.

os.path.join(*paths)

- This function joins one or more path components intelligently using the appropriate path separator for the current operating system.

- **import os**
- **path = os.path.abspath("relative/path/file.txt")**
- **print(path)**

os.path.abspath(path)

- This function returns the absolute version of a path, resolving any symbolic links and relative paths.
- import os
- path = os.path.abspath("relative/path/file.txt")
- print(path) # Output on Windows: C:\your\current\working\directory\relative\path\file.txt
- # Output on macOS/Linux: /your/current/working/directory/relative/path/file.txt

os.path.dirname(path)

- import os

- path = "/path/to/some/file.txt"
- directory = os.path.dirname(path)
- print(directory) # Output: /path/to/some

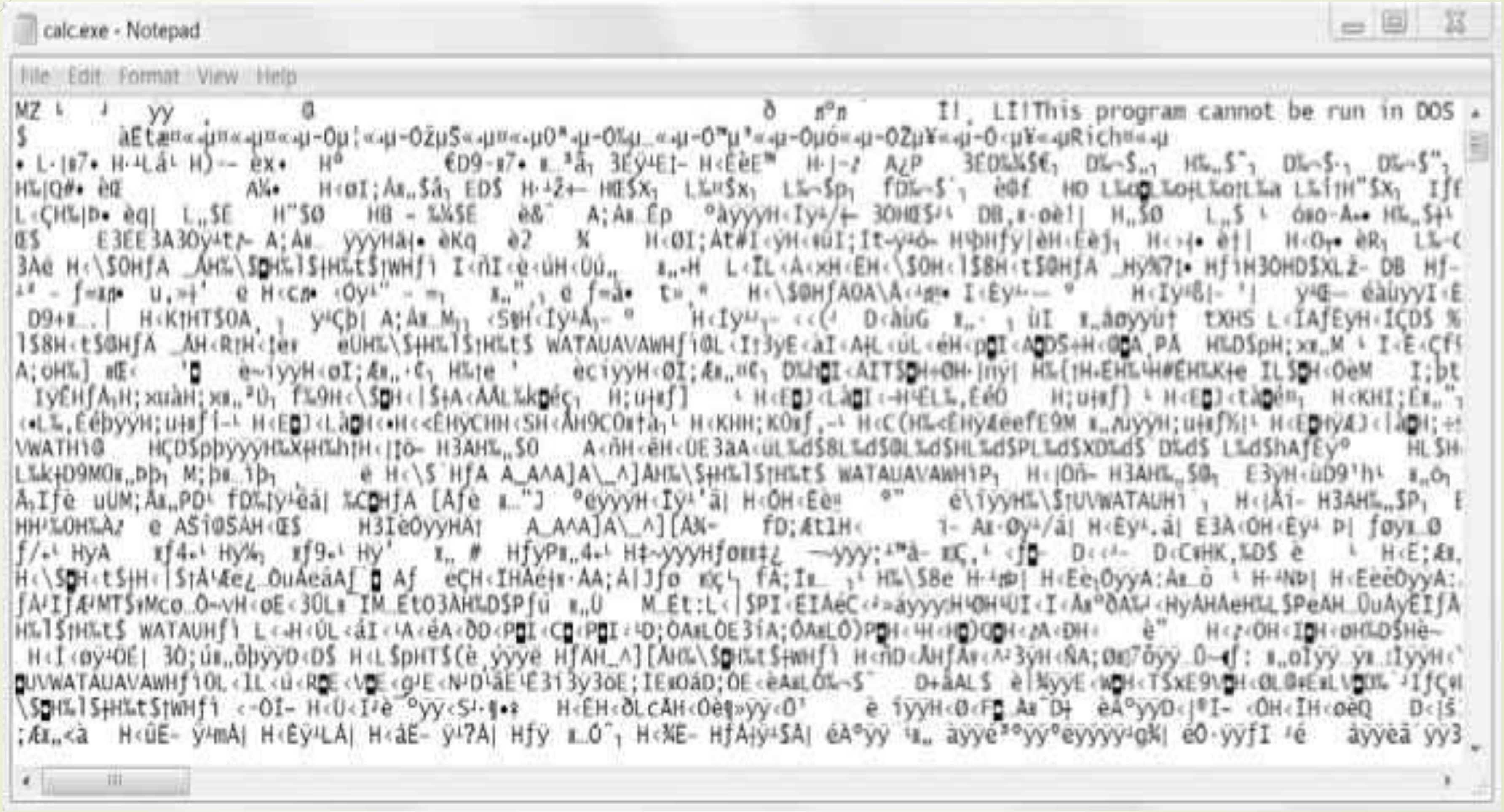
The File Reading/Writing Process

- Once you are comfortable working with folders and relative paths, you'll be able to specify the location of files to read and write.

- The functions covered in the next few sections will apply to plaintext files. *Plaintext files* contain only basic text characters and do not include font, size, or color information.

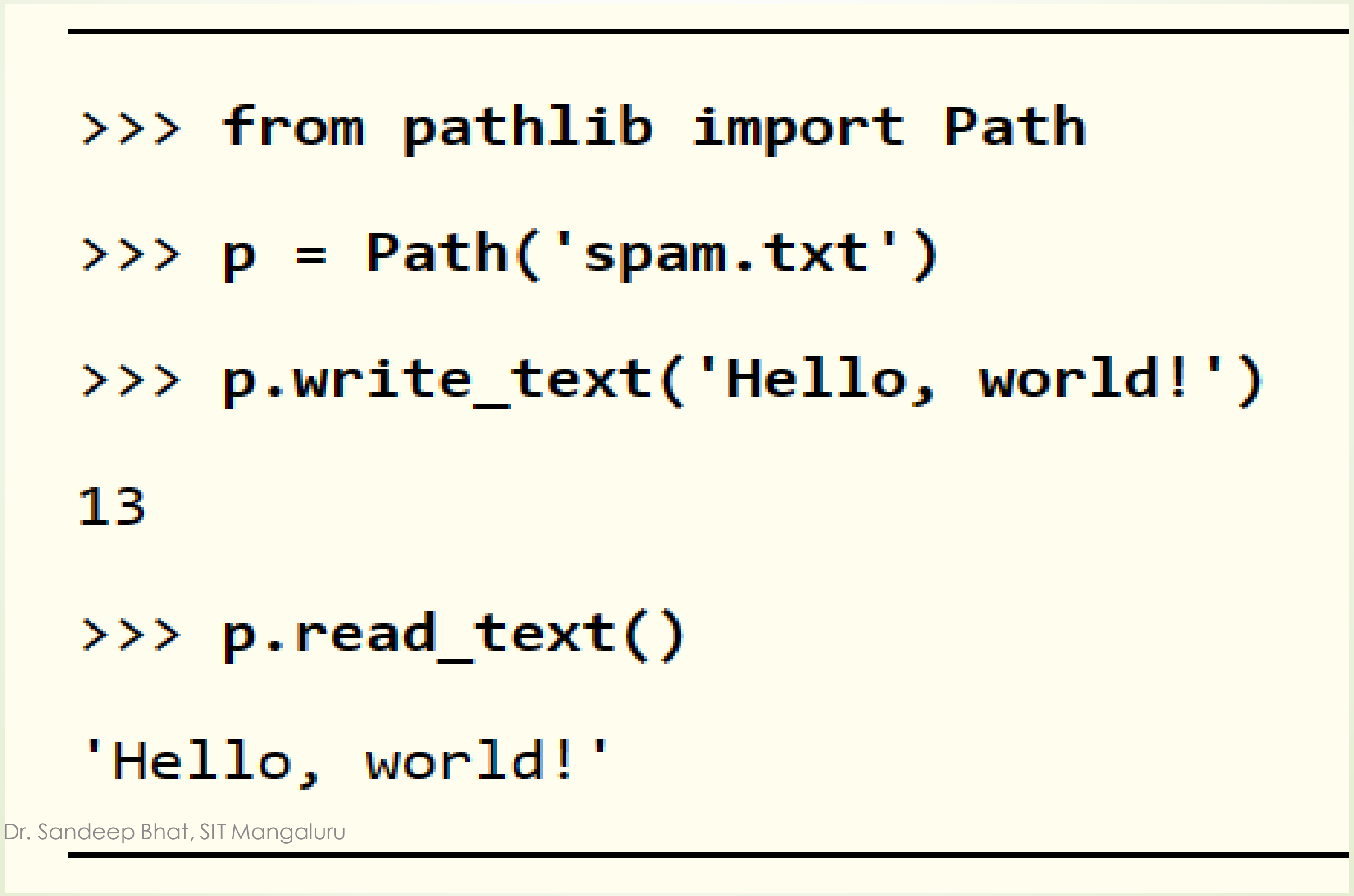
- Text files with the *.txt* extension or Python script files with the *.py* extension are examples of plaintext files.

- If you open a binary file in Notepad or TextEdit, it will look like scrambled nonsense, like in Figure 9-6.



```
MZ                                              !This program cannot be run in DOS
[illegible]
```

Eg.



```
>>> from pathlib import Path
>>> p = Path('spam.txt')
>>> p.write_text('Hello, world!')
13
>>> p.read_text()
'Hello, world!'
```

Cont..

- These method calls create a ***spam.txt*** file with the content 'Hello, world!'.

- The 13 that **write_text()** returns indicates that 13 characters were written to the file. (You can often disregard this information.)

- The **read_text()** call reads and returns the contents of our new file as a string: 'Hello, world!'.

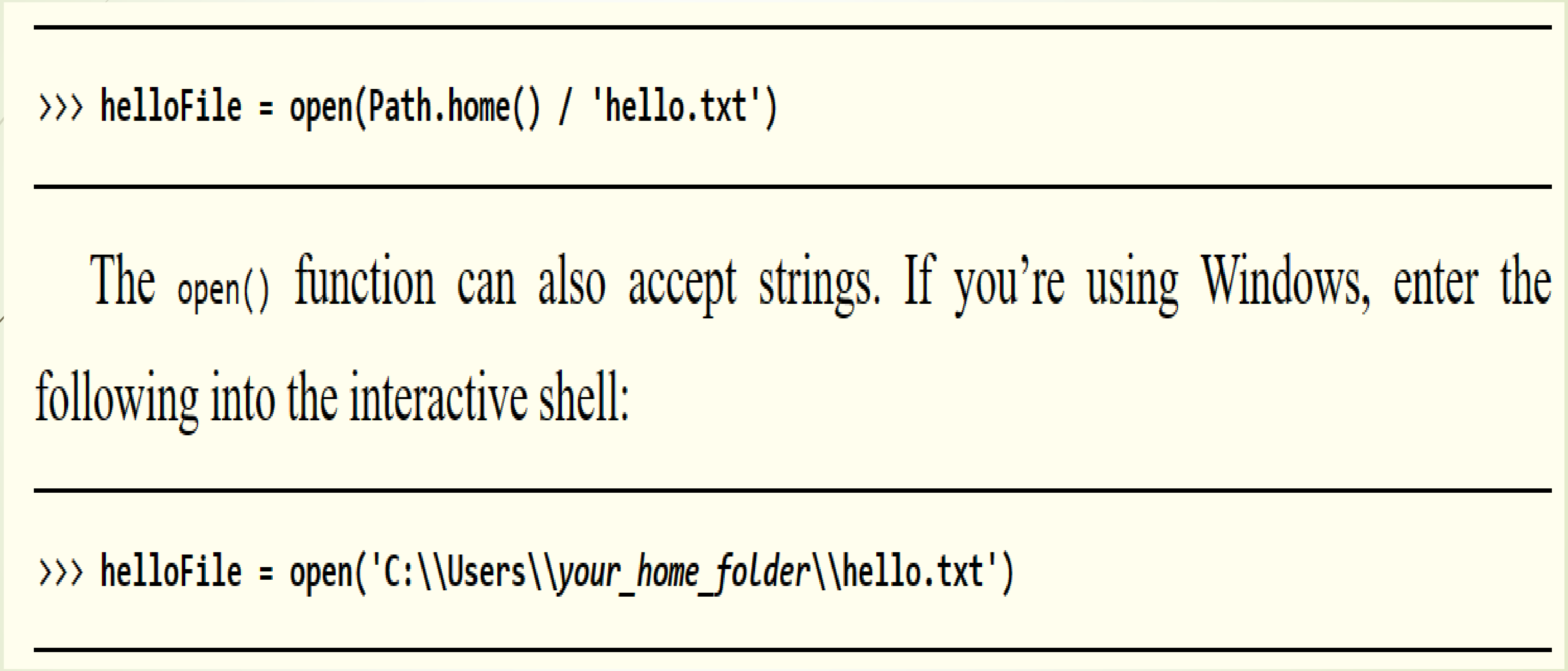
➢ There are three steps to reading or writing files in Python:

- **Call the open() function to return a File object.**
- **Call the read() or write() method on the File object.**
- **Close the file by calling the close() method on the File object.**

Opening Files with the open() Function

- To open a file with the open() function, you pass it a string path indicating the file you want to open; it can be either an absolute or relative path.
- The open() function returns a File object.
- Try it by creating a text file named *hello.txt* using Notepad or TextEdit.
- Type Hello, world! as the content of this text file and save it in your user home folder.

Eg.



```
>>> helloFile = open(Path.home() / 'hello.txt')
```

The `open()` function can also accept strings. If you're using Windows, enter the following into the interactive shell:

```
>>> helloFile = open('C:\\Users\\your_home_folder\\hello.txt')
```

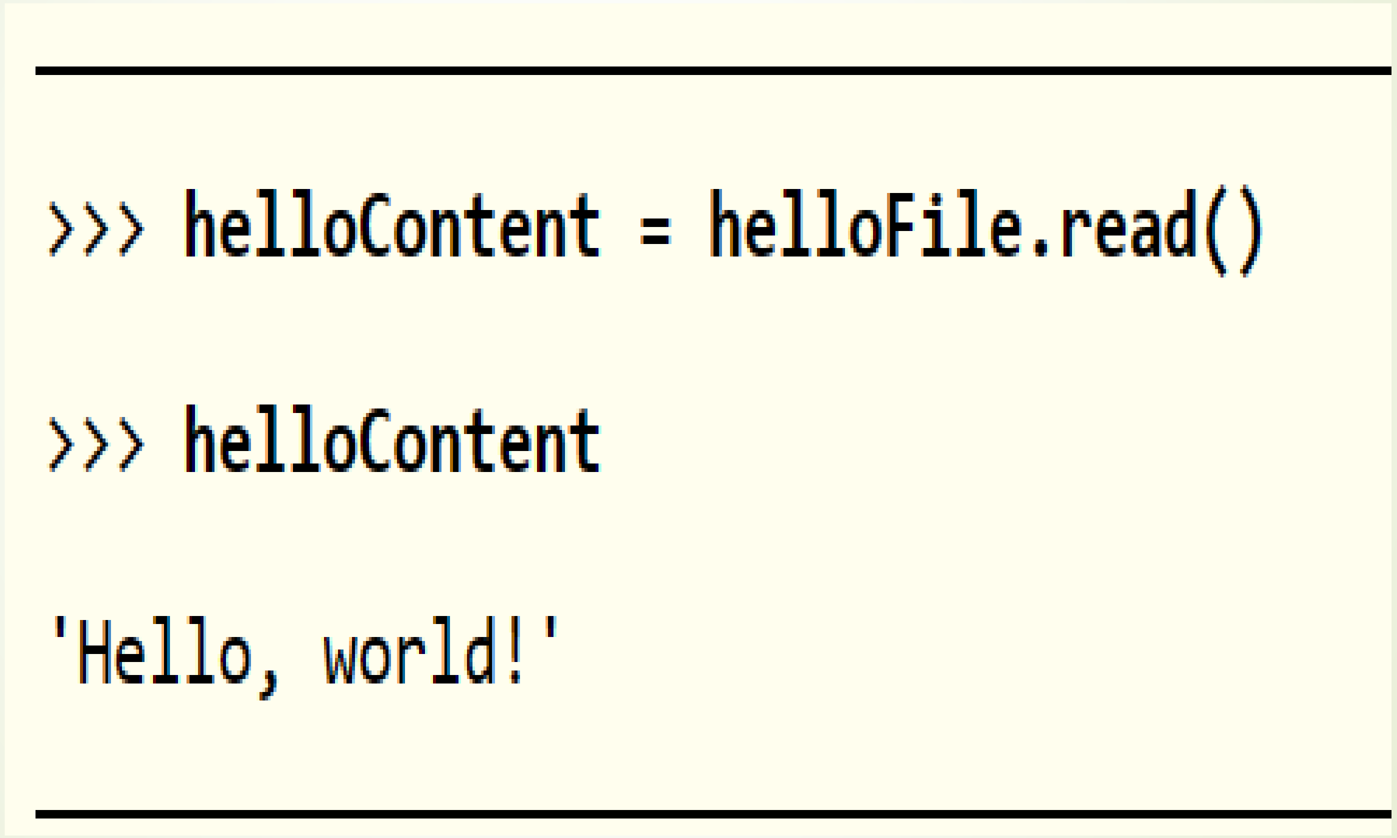
Reading the Contents of Files

- Now that you have a File object, you can start reading from it.

- If you want to read the entire contents of a file as a string value, use the File object's **read()** method.

- Let's continue with the ***hello.txt*** File object you stored in **helloFile**.

Eg.

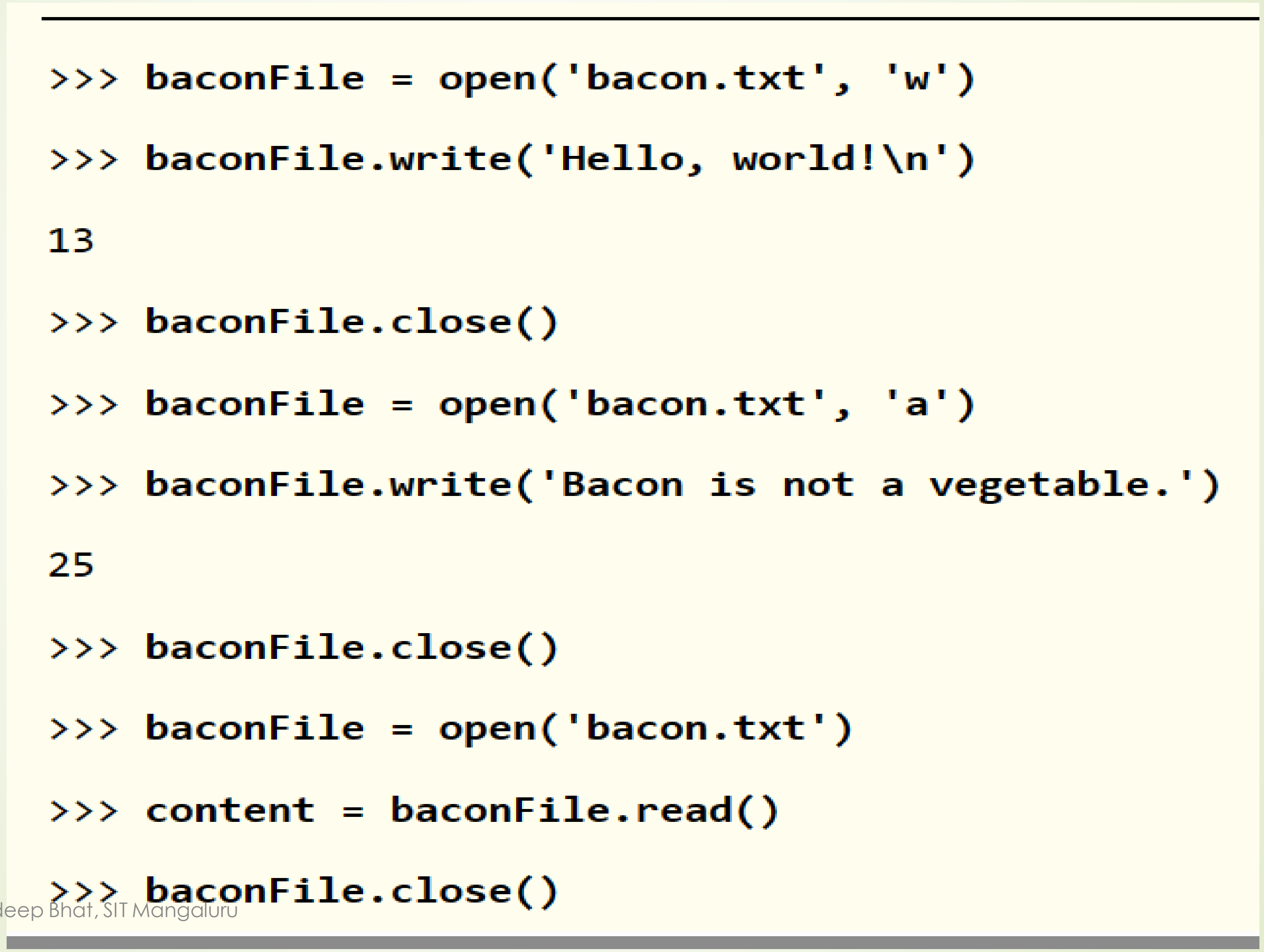


```
>>> helloContent = helloFile.read()
>>> helloContent
'Hello, world!'
```

Writing to Files

- **Python allows you to write content to a file in a way similar to how the print() function "writes" strings to the screen.**

- **You can't write to a file you've opened in read mode, though. Instead, you need to open it in "write plaintext" mode or "append plaintext" mode, or *write mode* and *append mode* for short.**



```
>>> baconFile = open('bacon.txt', 'w')
>>> baconFile.write('Hello, world!\n')
13
>>> baconFile.close()
>>> baconFile = open('bacon.txt', 'a')
>>> baconFile.write('Bacon is not a vegetable.')
25
>>> baconFile.close()
>>> baconFile = open('bacon.txt')
>>> content = baconFile.read()
>>> baconFile.close()
```

output

```
Hello, world!
Bacon is not a vegetable.
```

Cont..

- Write mode will overwrite the existing file and start from scratch, just like when you overwrite a variable's value with a new value.
- Pass 'w' as the second argument to open() to open the file in write mode. Append mode, on the other hand, will append text to the end of the existing file.
- Pass 'a' as the second argument to open() to open the file in append mode.

Saving Variables with the shelve Module

- You can save variables in your Python programs to binary shelf files using the shelve module.
- This way, your program can restore data to variables from the hard drive. The shelve module will let you add Save and Open features to your program.
- For example, if you ran a program and entered some configuration settings, you could save those settings to a shelf file and then have the program load them the next time it is run.

Eg.

```
>>> import shelve
>>> shelfFile = shelve.open('mydata')
>>> cats = ['Zophie', 'Pooka', 'Simon']
>>> shelfFile['cats'] = cats
>>> shelfFile.close()
```

Cont..

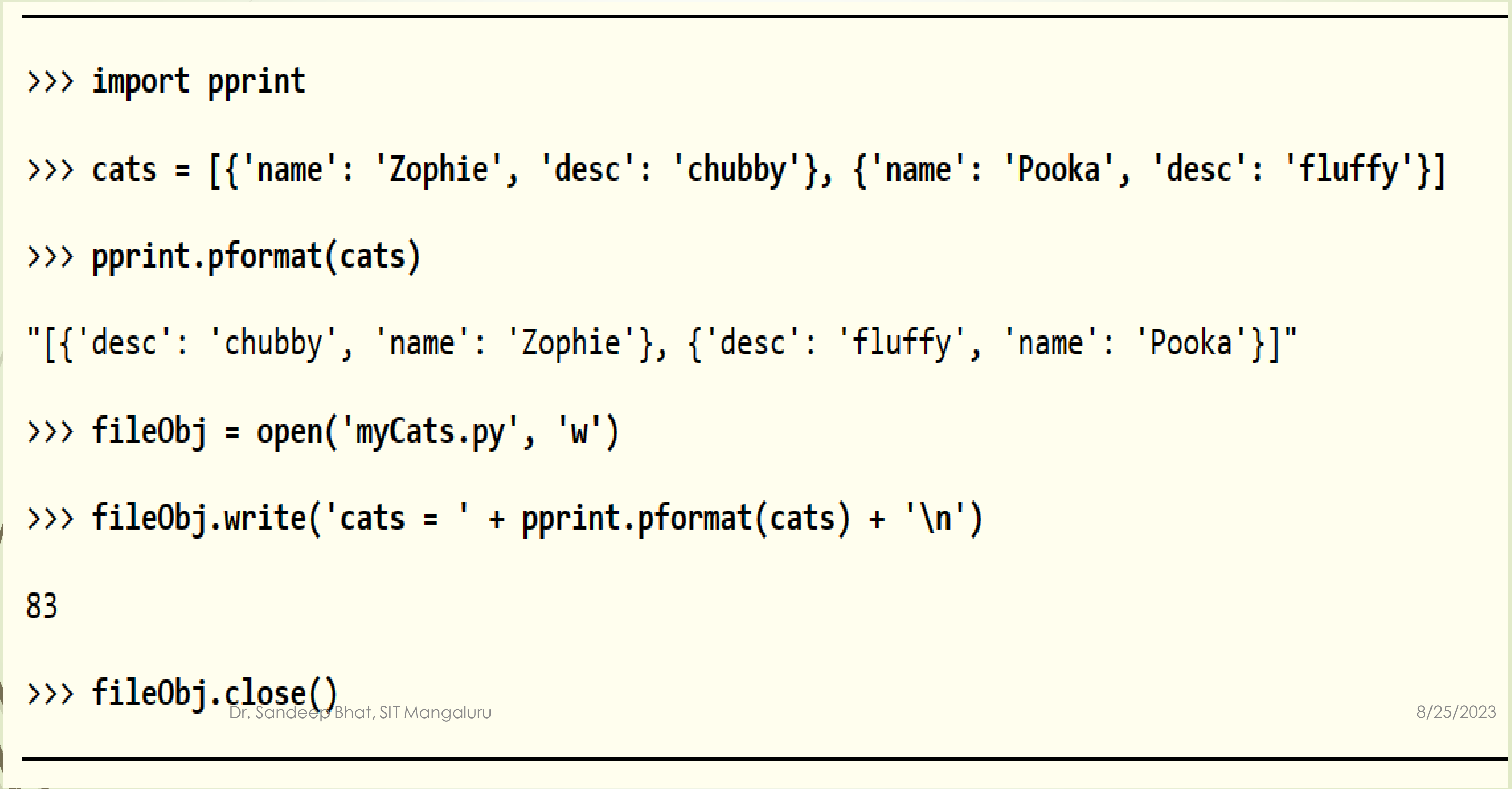
- To read and write data using the shelve module, you first import shelve.
- Call shelve.open() and pass it a filename, and then store the returned shelf value in a variable.
- You can make changes to the shelf value as if it were a dictionary. When you're done, call close() on the shelf value. Here, our shelf value is stored in shelfFile.
- We create a list cats and write shelfFile['cats'] = cats to store the list in shelfFile as a value associated with the key 'cats' (like in a dictionary). Then we call close() on shelfFile.

Saving Variables with the pprint.pformat() Function

- Recall from "Pretty Printing" that the pprint.pprint() function will "pretty print" the contents of a list or dictionary to the screen, while the pprint.pformat() function will return this same text as a string instead of printing it.

- Not only is this string formatted to be easy to read, but it is also syntactically correct Python code.

Eg.



```
>>> import pprint

>>> cats = [{'name': 'Zophie', 'desc': 'chubby'}, {'name': 'Pooka', 'desc': 'fluffy'}]

>>> pprint.pformat(cats)

"[{'desc': 'chubby', 'name': 'Zophie'}, {'desc': 'fluffy', 'name': 'Pooka'}]"

>>> fileObj = open('myCats.py', 'w')

>>> fileObj.write('cats = ' + pprint.pformat(cats) + '\n')

83

>>> fileObj.close()
```

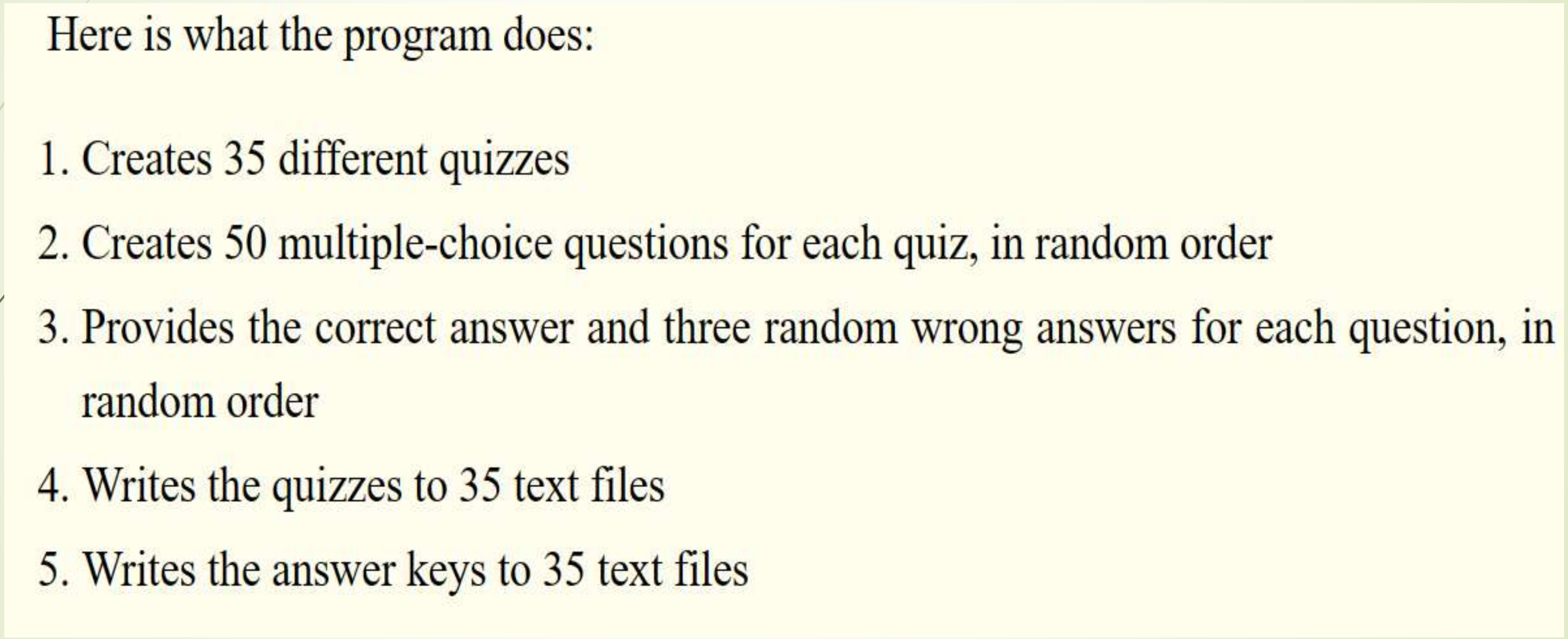
Cont..

- Here, we import pprint to let us use pprint.pformat().
- We have a list of dictionaries, stored in a variable cats.
- To keep the list in cats available even after we close the shell, we use pprint.pformat() to return it as a string.
- Once we have the data in cats as a string, it's easy to write the string to a file, which we'll call *myCats.py*.

Project: Generating Random Quiz Files

- Say you're a geography teacher with 35 students in your class and you want to give a pop quiz on US state capitals.
- Alas, your class has a few bad eggs in it, and you can't trust the students not to cheat.
- You'd like to randomize the order of questions so that each quiz is unique, making it impossible for anyone to crib answers from anyone else.
- Of course, doing this by hand would be a lengthy and boring affair.

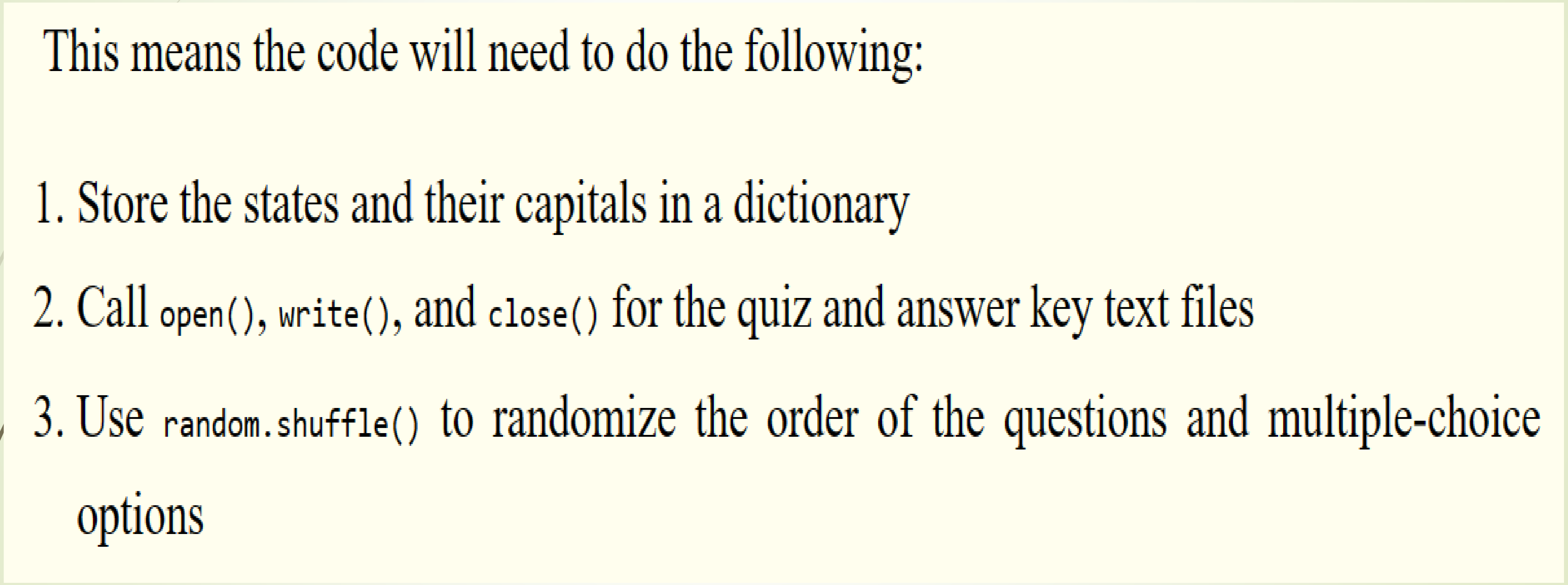
Cont..



Here is what the program does:

1. Creates 35 different quizzes
2. Creates 50 multiple-choice questions for each quiz, in random order
3. Provides the correct answer and three random wrong answers for each question, in random order
4. Writes the quizzes to 35 text files
5. Writes the answer keys to 35 text files

Cont..



This means the code will need to do the following:

1. Store the states and their capitals in a dictionary
2. Call `open()`, `write()`, and `close()` for the quiz and answer key text files
3. Use `random.shuffle()` to randomize the order of the questions and multiple-choice options

Step 1: Store the Quiz Data in a Dictionary

- **The first step is to create a skeleton script and fill it with your quiz data.**

- **Create a file named *randomQuizGenerator.py*, and make it look like the following:**

Step 2: Create the Quiz File and Shuffle the Question Order

- Now it's time to start filling in those TODOs.
- The code in the loop will be repeated 35 times—once for each quiz—so you have to worry about only one quiz at a time within the loop.
- First you'll create the actual quiz file. It needs to have a unique filename and should also have some kind of standard header in it, with places for the student to fill in a name, date, and class period.
- Then you'll need to get a list of states in randomized order, which can be used later to create the questions and answers for the quiz.

Step 3: Create the Answer Options

- Now you need to generate the answer options for each question, which will be multiple choice from A to D.

- You'll need to create another for loop—this one to generate the content for each of the 50 questions on the quiz.

- Then there will be a third for loop nested inside to generate the multiple-choice options for each question.

Step 4: Write Content to the Quiz and Answer Key Files

- **All that is left is to write the question to the quiz file and the answer to the answer key file.**

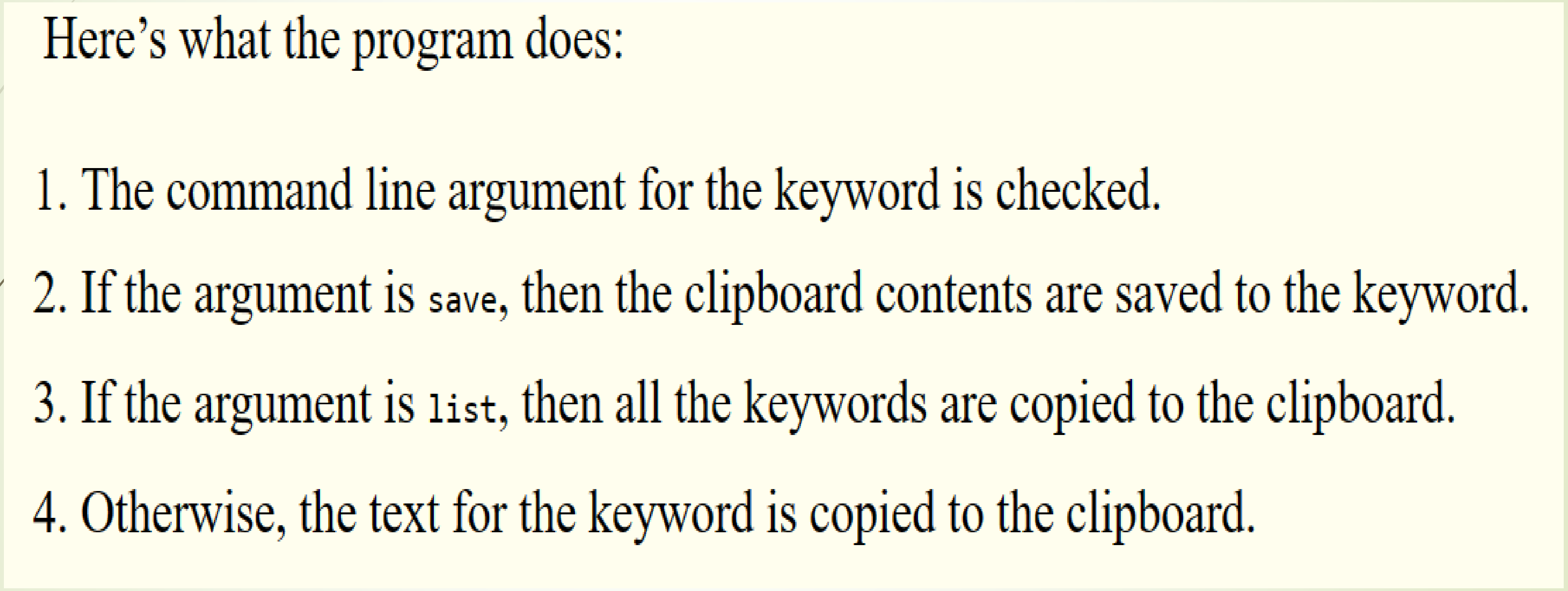
PROJECT: UPDATABLE MULTI-CLIPBOARD

- The program will save each piece of clipboard text under a keyword. For example, when you run py mcb.pyw save spam, the current contents of the clipboard will be saved with the keyword *spam*.

- This text can later be loaded to the clipboard again by running py mcb.pyw spam.

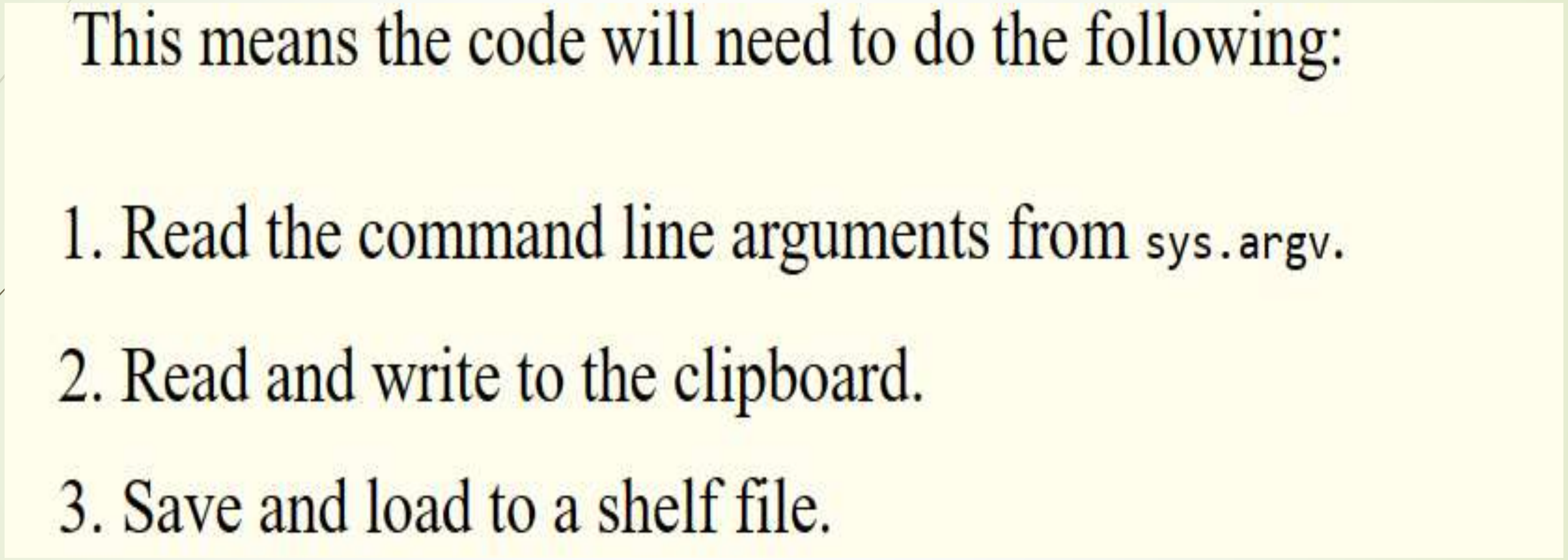
- And if the user forgets what keywords they have, they can run py mcb.pyw list to copy a list of all keywords to the clipboard.

Cont..


Here’s what the program does:

1. The command line argument for the keyword is checked.
2. If the argument is `save`, then the clipboard contents are saved to the keyword.
3. If the argument is `list`, then all the keywords are copied to the clipboard.
4. Otherwise, the text for the keyword is copied to the clipboard.

Cont..



This means the code will need to do the following:

1. Read the command line arguments from `sys.argv`.
2. Read and write to the clipboard.
3. Save and load to a shelf file.

Step 1: Comments and Shelf Setup

- Let’s start by making a skeleton script with some comments and basic setup.

Step 2: Save Clipboard Content with a Keyword

- **The program does different things depending on whether the user wants to save text to a keyword, load text into the clipboard, or list all the existing keywords.**

Step 3: List Keywords and Load a Keyword's Content

- **Finally, let's implement the two remaining cases: the user wants to load clipboard text in from a keyword, or they want a list of all available keywords.**