Week 9

Top-Down Design and File Output

Learning Objectives

Top-Down Design of Programs

- Create appropriately detailed top-down hierarchies when given a primary (complex) goal
- Explain and give examples of advantages and disadvantages of top-down design
- Define and create trees, roots, nodes, parent, children and leaves in this context
- Utilize a top-down design approach as part of creating a Python program

File Output

- Use ‘open’ command within Python to write external files
- Define and utilize file open designators (e.g., ‘r’, ‘w’, ‘a’)
- Write to files using write command

Hierarchies in practice

We see hierarchies like this all the time, to help us manage and organize

Universities:	University -> Academic/Non-Academic -> Colleges/Offices -> Departments -> Individuals
Companies:	President/CEO, Vice Presidents, Division Directors, Group Managers, Employees
Anatomy:	Body -> Systems -> Organs -> Tissues -> Cells
Communities:	Metro region -> Cities -> Zones/Neighborhoods -> Buildings
Sports Leagues:	League -> Conference -> Division

Top-Down Program Design

We'll use top-down design as a way of organizing many of our programs

- Break the problem into individual "large" steps
- Break those into smaller steps
- Stop once the code is "obvious" from the description
- Typically, this is once implementing a concept will take only a few lines of code (on the order of 1 - 10 lines of code)
- Can turn the nodes into comments to help show structure

Some Computing Terminology

Hierarchies like this in computing are usually called "Trees"

- The tree has a "root" at the base
- Individual elements are often called "nodes"
- Nodes have:
    - A "parent" (the node just above; the root has no parent)
    - Possibly "children" (the nodes that descend from it, below)
    - Nodes without children are called "leaves"

Hierarchies

The key is that a hierarchy helps **manage complexity**.

On balance, hierarchies are a useful way to organize ideas, processes, etc.

- It should be one of the first ways you think of organizing when approaching a complex problem

ENGINEERING
TEXAS A&M UNIVERSITY

Top-down design going forward

- Plan for this to be your go-to method of approaching programming problems in our class.
- We will see some alternatives soon.
- Many methods result in a hierarchy - the key for it being ‘top-down’ is that you start at the top and then work into smaller pieces.
- For this class, when requested to show a top-down plan, draw the hierarchy similar to what I’ve shown

ENGINEERING
TEXAS A&M UNIVERSITY

File Output

Writing to an external file

Input/Output

So far, we've only input and output from the PyCharm console

- The **print** command (for output)
- The **input** command (for input)

But, how do we deal with **files**?

File Extensions

Most file names have an “extension”

e.g., .pdf, .docx, .jpg, .mov, .mp3, .xlsx, .csv, etc.

The “extension” is just part of the name, it does **not necessarily** mean anything about what the file contains.

- You can rename any file with a different extension; the file isn’t changed.
- You can name your files created by Python anything, too (even things they are not)

Your computer and applications use the file extension to determine how to process the file, and with which application to open the file.

- But, it is not a guarantee.

File Name

If you want to make a file where the data inside matches the hint given by its extension, you need to write the data to the file in the right way.

- e.g. for a .pdf file, you must write binary data in the exact form expected for an Adobe Acrobat file

Many file extensions are associated with proprietary programs, or are very complicated to read/write

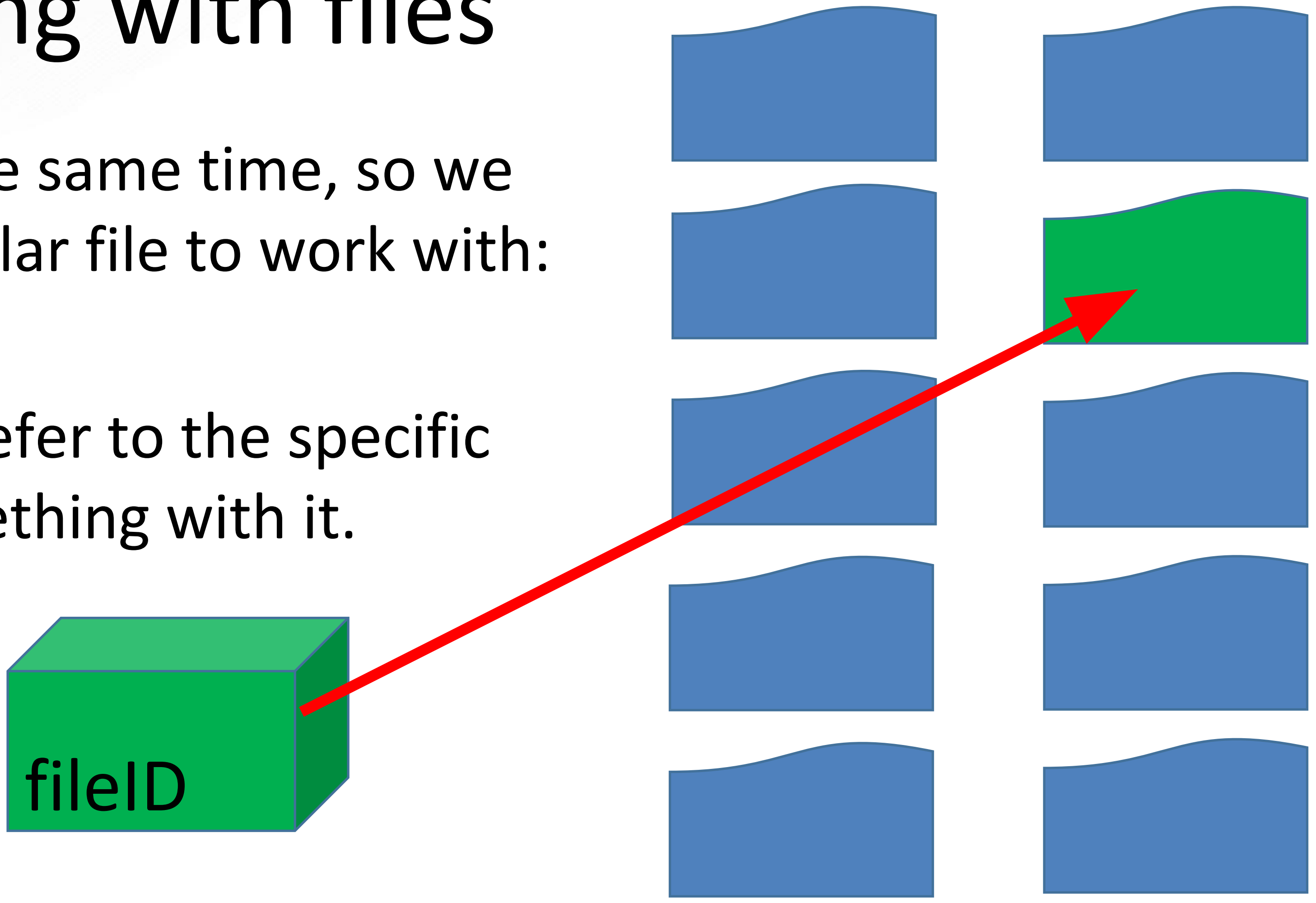
- There are sometimes libraries/modules you can use to help with this
- Most of the time you need to create the files using the right programs

The basics of dealing with files

We can work with many at the same time, so we need a way to select a particular file to work with:

- A **file identifier** (a variable)

We will use this identifier to refer to the specific file when we want to do something with it.



File Basics

We first “open” the file

- At this time, we associate an identifier with the file
- We need to specify how we will work with the file *(reading? writing? etc.)*

Then, we work with the file contents

- Reading/Writing

Finally, we “close” the file

Opening a file - one method

```
with open("<File Name>", "<designator>") as <fileID>:
```

Start with the command "with"

Opening a file

```
with open("<File Name>", "<designator>") as <fileID>:
```

Designators:	
`r`	Reading (we will read data from an existing file)
`w`	Writing (we will write data to a new file, or overwrite an existing file)
`a`	Appending (we will append data to an existing file)
`rb`, `wb`, `ab`	We will read/write/append BINARY data
`r+`	We will read from AND write to the file
<nothing>	If there is no mode designator, then 'r' is assumed

Opening a file

```
with open("<File Name>", "<designator>") as <fileID>:
```

Then the word “as”

ENGINEERING
TEXAS A&M UNIVERSITY

Opening a file

```
with open("<File Name>", "<designator>") as <fileID>:
```

file ID – a variable name that we will use to refer to this specific file

Opening a file

```
with open("<File Name>", "<designator>") as <fileID>:
    # Stuff to do while file is open
    # More stuff
```

And finally a colon, after which we indent the subsequent lines.

Opening a file

```
with open("<File Name>", "<designator>") as <fileID>:
    # Stuff to do while file is open
    # More stuff
# Now file is closed
```

- When you finish with the indented portion, the file is automatically closed.
- The fileID variable can be used to refer to the file within the indented portion of the code

Using designator ‘w’ to create new file

```
with open("Example_File.txt", "w") as datFile:
    # Stuff to do while file is open
```

- If `Example_File.txt` exists already, using “w” as the designator will erase everything in it and start writing like new

  Note: if you want to write additional information to the end of a file, use “a” for append instead

- If `Example_File.txt` does **not** exist already, the file will be created.

File operation: writing

To write, we will use the write command*:

```
<fileID>.write(<string to write>)
```

* We'll assume we are **not** using binary (just standard read/write)

ENGINEERING
TEXAS A&M UNIVERSITY

File operation: **writing**

To write, we will use the write command:

```
<fileID>.write(<string to write>)
```

Start with the file identifier (variable name) for a file that was opened for writing or appending

File operation: writing

To write, we will use the write command:

```
<fileID>.write(<string to write>)
```

Then, there is a period, followed by the word “write” and then parentheses.

File operation: writing

To write, we will use the write command:

```
<fileID>.write(<string to write>)
```

Inside the parentheses is the string to write

File operation: writing

To write, we will use the write command:

```
<fileID>.write(<string to write>)
```

Examples:

```
myfile.write("First Line")
output_string = 'Second Line'
myfile.write(output_string)
```

The write command vs. the print statement

Write only writes **strings**

- You must first convert numbers to a string before writing

Write only writes a **single string**

- You cannot output multiple
- So, there is obviously no “space” separating separate strings written

Write does not automatically **add a new line**

- if wanted, a new line character must be explicitly added
- triple-quotes with newlines included can be used as well

Example: writing

Next

```
with open("MyOutput.txt", 'w') as outfile:
    outfile.write("Testing the write command.\n")
    x = 987
    outfile.write("Here's a number: "+str(x)+'\n')
    outfile.write("And another number:")
    outfile.write(str(21))
    outfile.write("\n")
# done
```

Example: writing

Next

```
with open("MyOutput.txt", 'w') as outfile:
    outfile.write("Testing the write command.\n")
    x = 987
    outfile.write("Here's a number: "+str(x)+'\n')
    outfile.write("And another number:")
    outfile.write(str(21))
    outfile.write("\n")
# done
```

The file MyOutput.txt is created and opened for writing. The next thing written will be at the beginning of the file

MyOutput.txt

Write

Example: writing

```
with open("MyOutput.txt", 'w') as outfile:
    outfile.write("Testing the write command.\n")
    x = 987
    outfile.write("Here's a number: "+str(x)+'\n')
    outfile.write("And another number:")
    outfile.write(str(21))
    outfile.write("\n")
# done
```

Next

The first line is written to the file. The newline at the end means the next write will begin on the next line of the file

MyOutput.txt

```
Testing the write command.
```

Write

Example: writing

```
with open("MyOutput.txt", 'w') as outfile:
    outfile.write("Testing the write command.\n")
    x = 987
    outfile.write("Here's a number: "+str(x)+'\n')
    outfile.write("And another number:")
    outfile.write(str(21))
    outfile.write("\n")
# done
```

Next

x is created in memory, holding the value 987

MyOutput.txt

```
Testing the write command.
```

Write

Example: writing

```
with open("MyOutput.txt", 'w') as outfile:
    outfile.write("Testing the write command.\n")
    x = 987
    outfile.write("Here's a number: "+str(x)+'\n')
    outfile.write("And another number:")
    outfile.write(str(21))
    outfile.write("\n")
# done
```

Next

The next line is written, with a newline. Notice that the variable was converted to a string.

MyOutput.txt

```
Testing the write command.
Here's a number: 987
```

Write

Example: writing

```
with open("MyOutput.txt", 'w') as outfile:
    outfile.write("Testing the write command.\n")
    x = 987
    outfile.write("Here's a number: "+str(x)+'\n')
    outfile.write("And another number:")
    outfile.write(str(21))
    outfile.write("\n")
# done
```

Next

Another string is written, but not a newline, so the next thing will appear right afterward.

MyOutput.txt

```
Testing the write command.
Here's a number: 987
And another number:
```

Write

Example: writing

```
with open("MyOutput.txt", 'w') as outfile:
    outfile.write("Testing the write command.\n")
    x = 987
    outfile.write("Here's a number: "+str(x)+'\n')
    outfile.write("And another number:")
    outfile.write(str(21))
    outfile.write("\n")
# done
```

Next

The number 21 is converted to a string and output, again with no newline.

MyOutput.txt

```
Testing the write command.
Here's a number: 987
And another number:21
```

Write

Example: writing

```
with open("MyOutput.txt", 'w') as outfile:
    outfile.write("Testing the write command.\n")
    x = 987
    outfile.write("Here's a number: "+str(x)+'\n')
    outfile.write("And another number:")
    outfile.write(str(21))
    outfile.write("\n")
# done
```

Next

Now a newline is written.

MyOutput.txt

```
Testing the write command.
Here's a number: 987
And another number:21
```

Write

Example: writing

```
with open("MyOutput.txt", 'w') as outfile:
    outfile.write("Testing the write command.\n")
    x = 987
    outfile.write("Here's a number: "+str(x)+'\n')
    outfile.write("And another number:")
    outfile.write(str(21))
    outfile.write("\n")
# done
```

The file is closed – nothing more can be written.

MyOutput.txt

```
Testing the write command.
Here's a number: 987
And another number:21
```

More to come next week:

- We’ll look at another way of opening files
  - More flexible, but requires you to close the file
- We’ll start reading from files
- We’ll work with comma separated value (CSV) files
- We’ll look at how to work with files in another folder on your computer