File output and Exception Handling

Python and files

- Using files is a 3-step process
  1. Open the file in the desired mode
  2. Use the file
  3. Close the file

- Think about file input and output (file I/O) as a series of ***pipes***
  - Pipes establish a connection between two things
  - Pipes have directionality

2 ways to do the same thing

```
# Displays every line in the file
fileIn = open("story.txt", "r")
for line in fileIn:
    line = line.strip()
    print(line)
fileIn.close()

# Also displays every line in the file
with open("story.txt", "r") as fileIn:
    for line in fileIn:
        print(line.strip())
```

About using “with”

- The **`with`** keyword implicitly calls the `open` and `close` on the file

- It can work most of the time for basic file input and output, but not always

- The 3-step process will ALWAYS work

About output files

- We'll focus on text files that are human-readable
- File formats are usually
  - .txt
  - .csv
- Eventually we'll cover binary object files (called pickling)

A program that writes to a file

```
# 1. Open the file for reading
fileOut = open("output.txt", "w")

# 2. Write to the file
print("Hello y'all!", file=fileOut)

# 3. Close the file
fileOut.close()
```

Opening a file for writing

- The `open` function creates the file object

- Depending on the mode given – it will either be a read “pipe” or a write “pipe” and either for text or binary data

- Remember file objects (or “pipes”) have a direction
- We want to write FROM a program INTO a file

File output caveat

- By default, if a file already exists by the name we specify, it will be overwritten

- So, in our case, if we already have a file named “output.txt”, we will lose all its old contents

- There’s no warning, so be careful not to open a file you don’t want to lose!

- So don’t do this:

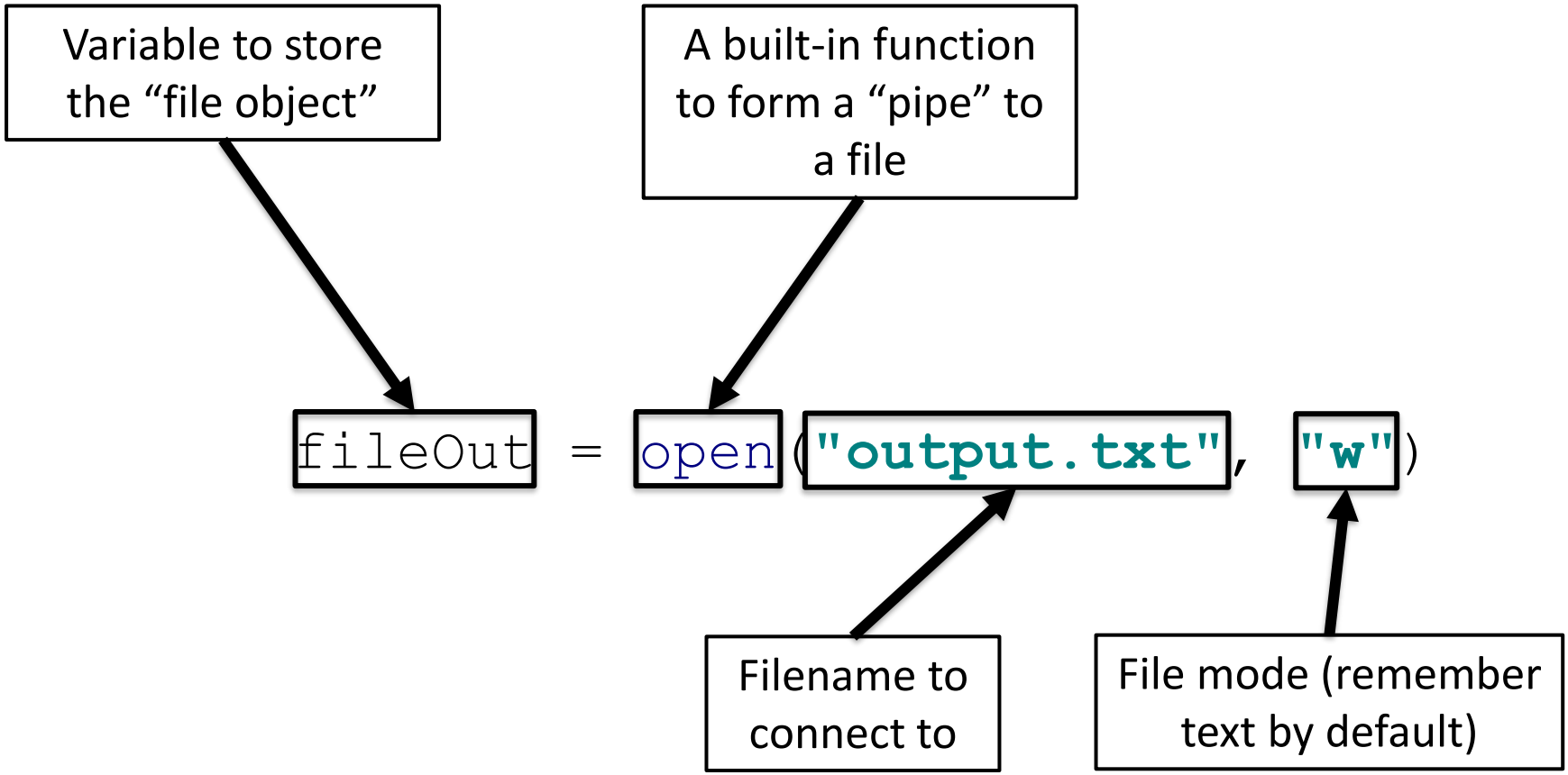
```
fileOut = open("myLifesWork.py", "w")
```

File access modes

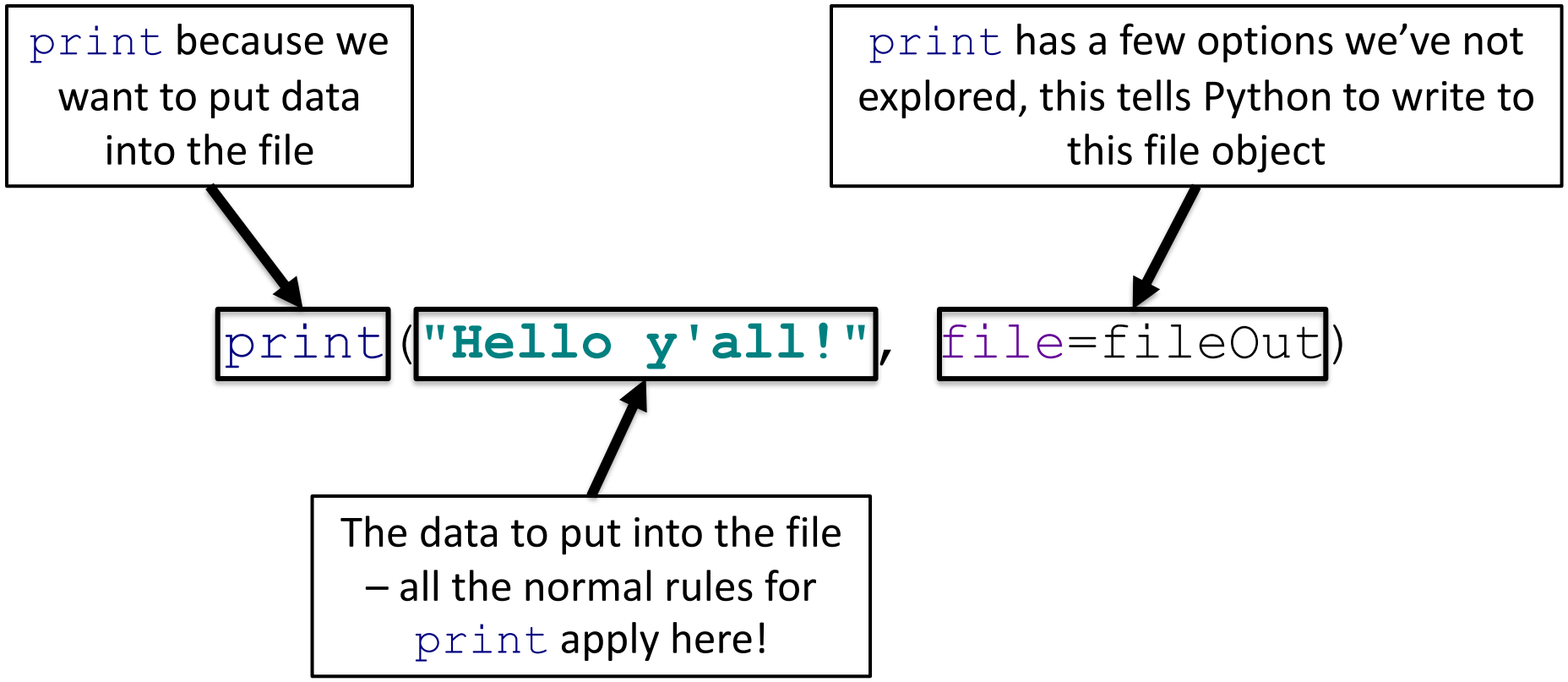
- Other modes exist, this isn't comprehensive!

Mode	Description
"tr"	Read from a text file (the "t" is optional) If the file doesn't exist, Python generates an error
"tw"	Write to a text file (the "t" is optional) If the file exists, its contents are overwritten If the file doesn't exist, it is created
"ta"	Append to a text file (the "t" is optional) If the file exists, new data is appended to it If the file doesn't exist, it is created
"br"	Read from a binary file (the "b" is required) If the file doesn't exist, Python generates an error
"bw"	Write to a binary file (the "b" is required) If the file exists, its contents are overwritten If the file doesn't exist, it is created
"ba"	Append to a binary file (the "b" is required) If the file exists, new data is appended to it If the file doesn't exist, it is created

Opening a file for writing syntax



Writing to a file syntax



- 1st parameter is still our normal output
- 2nd parameter is a ***named*** parameter (the file object to write to)

Writing to a file

- Because we're using `print` I had to form a single string, but commas and formatting still works!

- Because we're using the `print` function a newline (`"\n"`) is added automatically for us (but we can change that behavior)
  - Remember: `print("Hi", end="")`

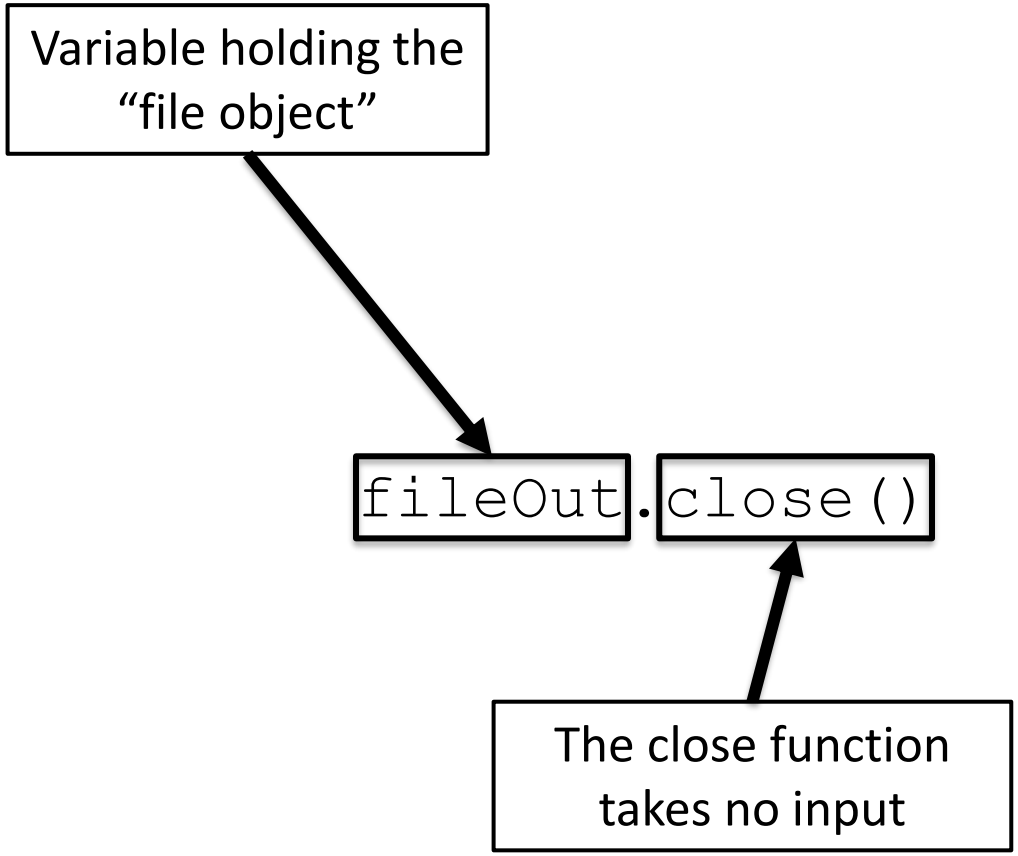
- Other file object functions exist to write character by character

- We'll get to the details of objects next week

Closing a file

- **After** you are done writing to a file, close it
- This closes the “pipe” and prevents corruption
- It’s a BadIdea™ to try moving or otherwise manipulating the file until it’s closed
- If your program crashes or otherwise ends it will automatically close the file

Closing a file syntax



What are the results of this program?

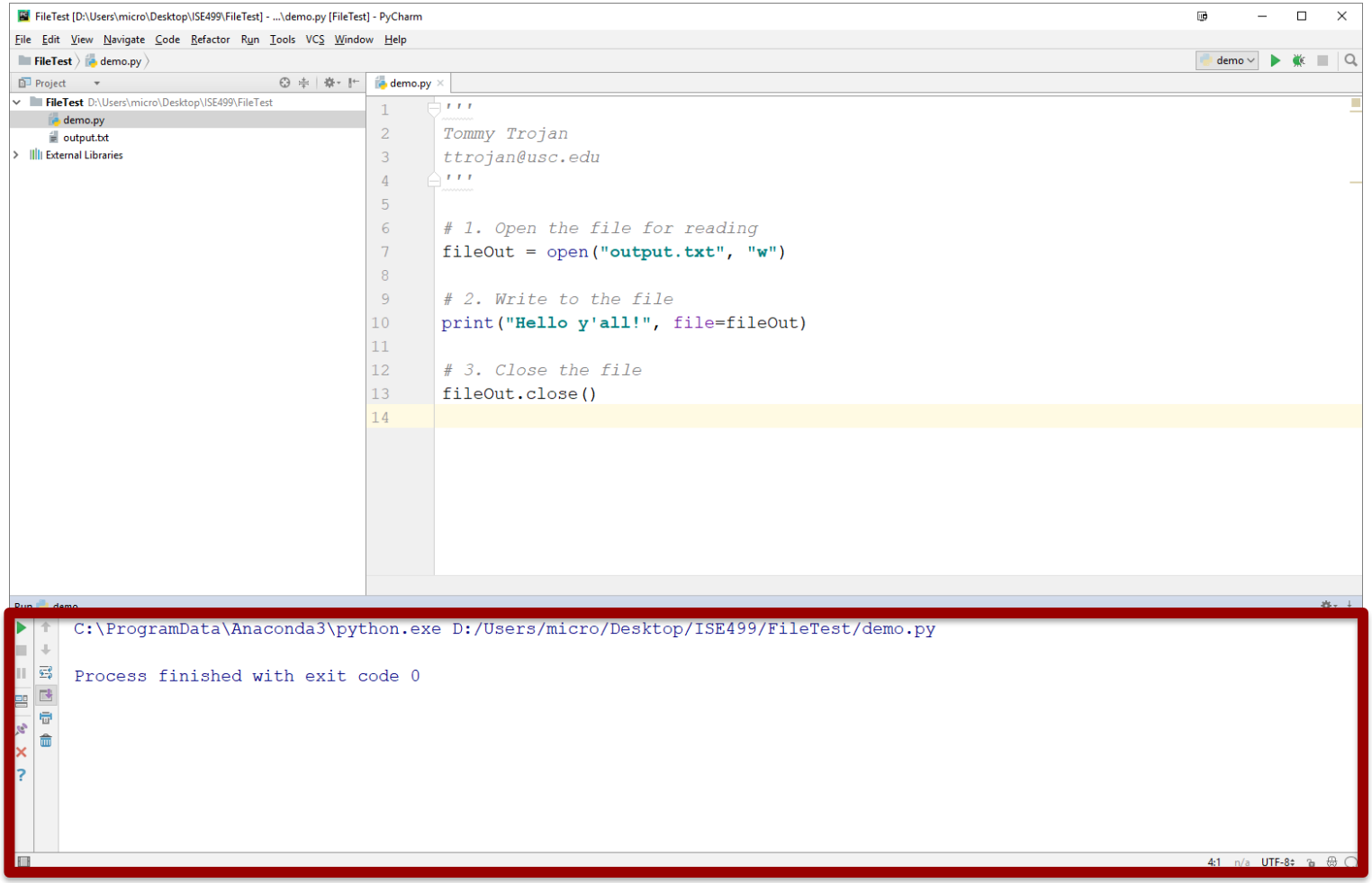
```
# 1. Open the file for writing
fileOut = open("output.txt", "w")

# 2. Write to the file
print("Hello y'all!", file=fileOut)

# 3. Close the file
fileOut.close()
```

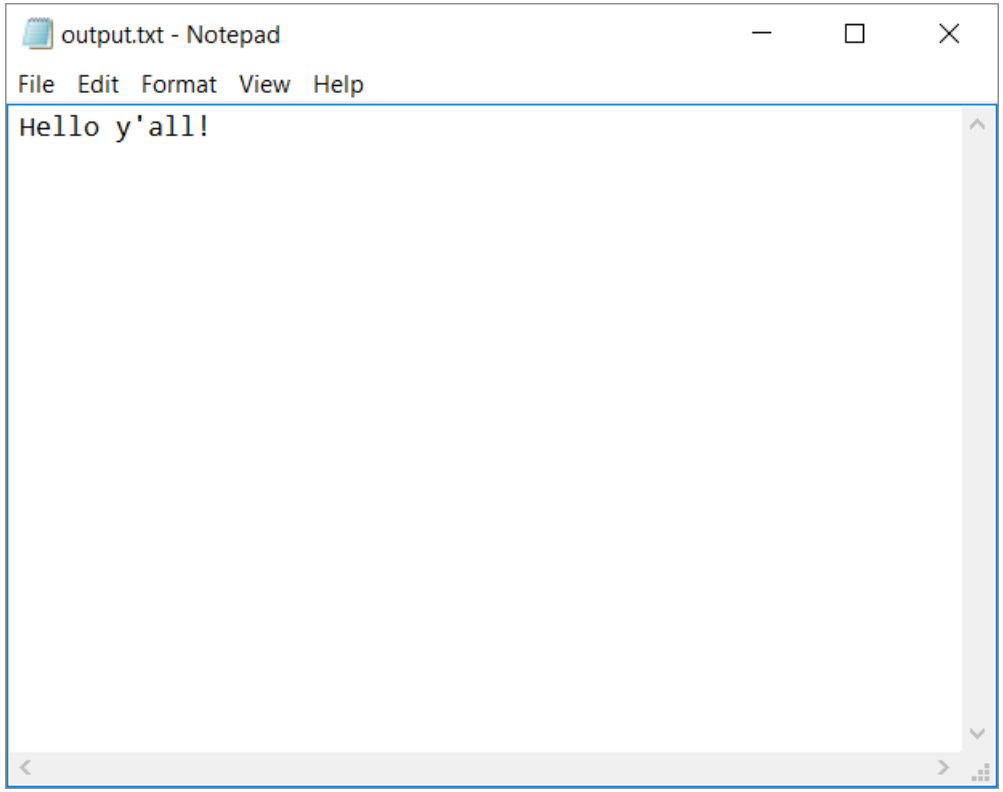
Example output

- If we run that program, we won't see any output to the console…



Example output

- But if we find the file named “output.txt”, it will have output!

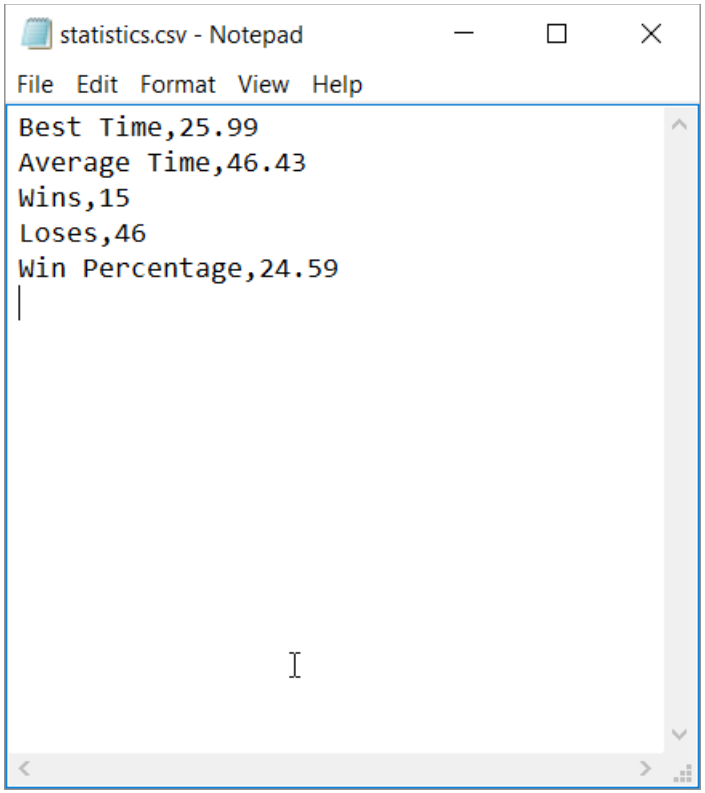


Tangent: viewing text files

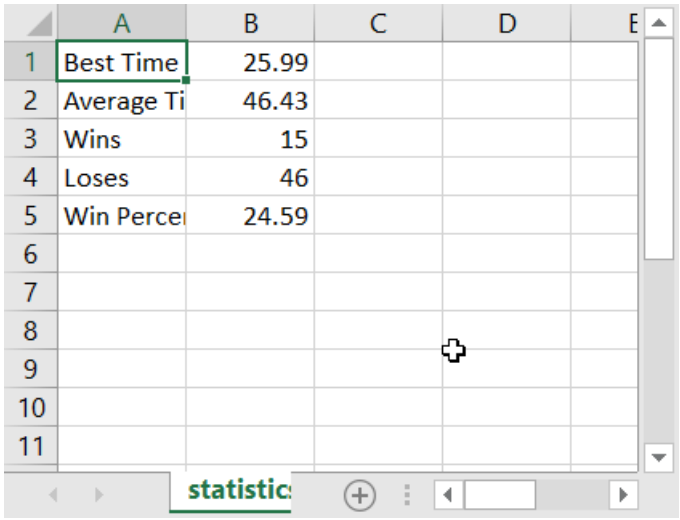
- Some files we'll use have suffixes that are tied to other programs
- Be sure to open these files with a text editor!
  - Mac: Sublime
  - Windows: Notepad
- For example, Microsoft Excel will attempt to open CSV files (comma separated values) files when double clicked

Tangent: a CSV in 2 programs

- What your computer sees:



- What your computer shows:



Comma Separated Values Files

- CSV files can represent a series of data fields, like a spreadsheet
- Files have a ".csv" suffix
  - Like "grades.csv" or "data.csv"
- CSV files are text files
  - They can be read on any platform
  - They can be read by any program
  - They usually have a header to indicate the different fields (but not always)
- Python has built-in libraries to manipulate CSV files

Representing data in a CSV

- Each line in a CSV file represents one line in a table or spreadsheet
- Commas separate each column within each line
- Often the first line is the headers (but not necessarily)

CSV Example

DEPT	COURSE_NUMBER	SEMESTER	NUMBER_OF_STUDENTS
ISE	150	Spring19	30
BUAD	101	Spring19	40
ISE	310	Spring19	35

CSV Example

DEPT	COURSE_NUMBER	SEMESTER	NUMBER_OF_STUDENTS
ISE	150	Spring19	30
BUAD	101	Spring19	40
ISE	310	Spring19	35

class.csv

```
DEPT,COURSE_NUMBER,SEMESTER,NUMBER_OF_STUDENTS
```

CSV Example

DEPT	COURSE_NUMBER	SEMESTER	NUMBER_OF_STUDENTS
ISE	150	Spring19	30
BUAD	101	Spring19	40
ISE	310	Spring19	35

class.csv

```
DEPT,COURSE_NUMBER,SEMESTER,NUMBER_OF_STUDENTS
ISE,150,Spring19,30
```

CSV Example

DEPT	COURSE_NUMBER	SEMESTER	NUMBER_OF_STUDENTS
ISE	150	Spring19	30
BUAD	101	Spring19	40
ISE	310	Spring19	35

class.csv

```
DEPT,COURSE_NUMBER,SEMESTER,NUMBER_OF_STUDENTS
ISE,150,Spring19,30
BUAD,101,Spring19,40
```

CSV Example

DEPT	COURSE_NUMBER	SEMESTER	NUMBER_OF_STUDENTS
ISE	150	Spring19	30
BUAD	101	Spring19	40
ISE	310	Spring19	35

class.csv

```
DEPT,COURSE_NUMBER,SEMESTER,NUMBER_OF_STUDENTS
ISE,150,Spring19,30
BUAD,101,Spring19,40
ISE,310,Spring19,35
```

Processing CSV files

- CSV files can be processed by Python libraries

- But require a few conditions
  1. The document needs to be “well formed” – the library can’t handle errors
  2. The header row must exist

- Otherwise, you can use the `split` string function

Processing CSV example

class.csv:

```
DEPT,COURSE_NUMBER,SEMESTER,NUMBER_OF_STUDENTS
ISE,150,Spring19,30
BUAD,101,Spring19,40
ISE,310,Spring19,35
```

```
with open("class.csv", "r") as csvFile:
    # Skip the header line
    csvFile.readline()
    for line in csvFile:
        row = line.split(",")
        print("Course: {}-{}".format(row[0], row[1]))
```

```
Course: ISE-150
Course: BUAD-101
Course: ISE-310
```

Processing CSV files

- Notice that you can use `readline` to skip the header line

- Python has libraries to handle CSV files
- To use the library **`import`** the CSV library to get access to its functions

- The object `reader` can turn each line into a list for you

Using the CSV library

```
import csv

# Create a reader that will handle the CSV
# It must be linked to the file
classReader = csv.reader(open("class.csv"))

# Skip the header line
next(classReader)

# read and display the data
for row in classReader:
    print(row)
```

```
['ISE', '150', 'Spring19', '30']
['BUAD', '101', 'Spring19', '40']
['ISE', '310', 'Spring19', '35']
```

Errors and exceptions

Revisiting the input function

- Input ***always*** returns a string – even if you want something else

- So far, we've been casting a string, but that doesn't always work…

```
age = int(input("Enter your age: "))
```

Revisiting the input function

- Input ***always*** returns a string – even if you want something else

- So far we've been casting a string, but that doesn't always work…

```
age = int(input("Enter your age: "))
```

```
Enter your age: Nineteen
Traceback (most recent call last):
  File "D:/Demo01/test01.py", line 10, in <module>
    age = int(input("Enter your age: "))
ValueError: invalid literal for int() with base 10: 'nineteen'
```

Checking a string's contents

- Python provides many ways to check a string's contents
  - Using the `in` keyword
  - Analyzing character by character
  - Regular expressions
  - String's built-in functions

- Let's use some of string's built in functions to prevent casting problems

Checking a string's contents

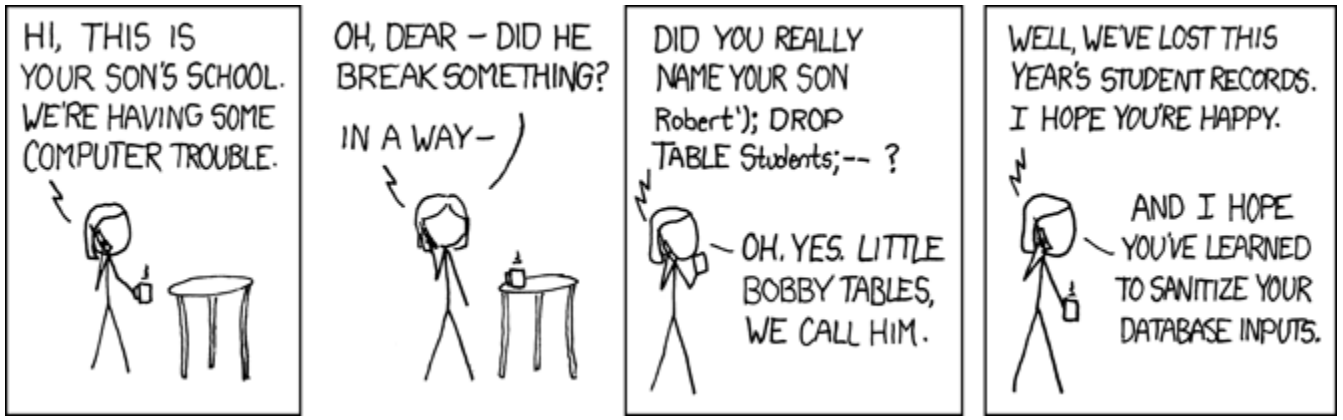
- Any string can do the following

Function	Description
`uInput.isalnum()`	Returns **`True`** if **`uInput`** contains only letters and numbers Returns **`False`** otherwise
`uInput.isalpha()`	Returns **`True`** if **`uInput`** contains only letters Returns **`False`** otherwise
`uInput.isdigit()`	Returns **`True`** if **`uInput`** contains only digits Returns **`False`** otherwise
`uInput.isspace()`	Returns **`True`** if **`uInput`** contains only whitespace Returns **`False`** otherwise
`uInput.isprintable()`	Returns **`True`** if **`uInput`** contains only printable characters Returns **`False`** otherwise

Checking input

- It's a GoodIdea™ to check the user's input before processing it

- Programmer's Rule #1: Never trust user input



https://xkcd.com/327/

- Some call it "programming defensively"
- We'll just call it "required" from now on

Checking input

- We can use the string functions to check the user's input

```
ageStr = input("Enter your age: ")
while not ageStr.isdigit():
    ageStr = input("Enter a number for your age: ")
age = int(ageStr)
print("You're {}".format(age))
```

```
Enter your age: nineteen
Enter a number for your age: twenty
Enter a number for your age: 40
You're 40
```

But what about…

- Are there any potential problems here?

```
numStr = input("Enter a numerator: ")
while not numStr.isdigit():
    numStr = input("Enter a number for the numerator: ")
num = int(numStr)

denStr = input("Enter a denominator: ")
while not denStr.isdigit():
    denStr = input("Enter a number for the denominator: ")
den = int(denStr)

result = num/den
```

Nothing is bullet proof

- Even after checking for numbers

```
result = num/den
```

- Could still cause a problem…

```
Enter a numerator: 11
Enter a denominator: 0
Traceback (most recent call last):
  File "D:/Demo01/test01.py", line 20, in <module>
    result = num/den
ZeroDivisionError: division by zero
```

Exceptions in Python

- An exception is an anomalous condition that can require special processing (or exception handling) to recover from
- Exceptions are usually errors, but they don’t have to be
- Python communicates all errors as exceptions
- But every exception is not necessarily an error

Exception handling

- When Python runs into an error, it generates an exception

- If nothing is done with the exception, Python halts what it's doing (running your program) and displays an error message

- Most basic way to handle (or trap) exceptions is to use the **`try`** statement with an **`except`** clause

What exactly happens here…

```
result = num/den
```

- When Python sees the code…
  1. It performs the value substitutions (`num` = 11, `den` = 0)
  2. Attempts the division
  3. Gets an error
  4. Displays the exception
  5. Stops the program

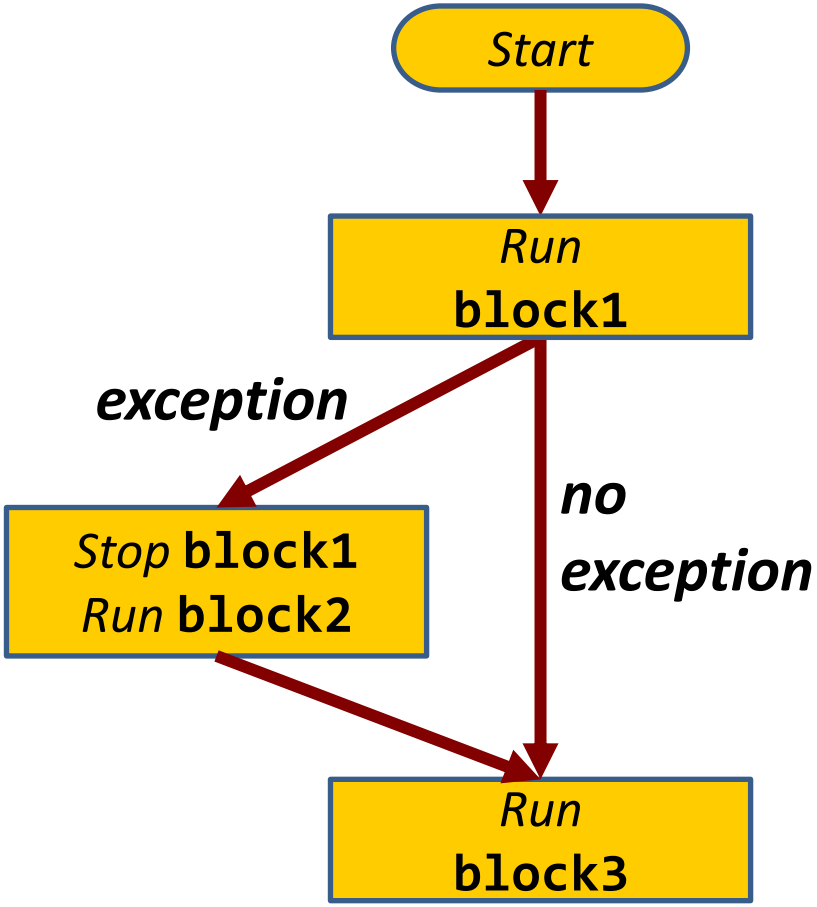
- We can interrupt the flow at step 4 by “handling” the exception

Program flow with an if statement visualized

```
try:
     block1

except:
     block2

block3
```



Common exceptions

- This list is not complete

Exception Type	Description
`IOError`	Raised when an I/O operation fails, such as when an attempt is made to open a nonexistent file in read mode.
`IndexError`	Raised when a sequence is indexed with a number of a nonexistent element.
`KeyError`	Raised when a dictionary key is not found.
`NameError`	Raised when a name (of a variable or function, for example) is not found.
`SyntaxError`	Raised when a syntax error is encountered.
`TypeError`	Raised when a built-in operation or function is applied to an object of inappropriate type.
`ValueError`	Raised when a built-in operation or function receives an argument that has the right type but an inappropriate value.
`ZeroDivisionError`	Raised when the second argument of a division or modulo operation is zero.

About exception handling

- When handling exceptions you can…
  1. Distinguish between and handle each type of exception separately
  2. Direct Python to execute a block of code for each type (or combination) of exception situations

- Exceptions are objects!
  - They have features and abilities
  - They can be specified and queried for information
  - (More on objects next week!)

Handling the exception

- Use the **`except`** keyword with a specific exception
  - Will run the following code block if that exact exception occurs
- Use the **`except`** keyword with several exceptions
  - Will run the following code block if ANY of the indicated exceptions occur
- Use the **`except`** keyword by itself for any exception
  - Will run the following code block if ANY exception occurs
- It is common to tier the exception handling
  - Most specific cases first
  - The general case at the end

Using exceptions

```
try:
    num = float(input("Numerator: "))
    den = float(input("Denominator: "))
    result = num/den
    print(result)
```

This may generate an exception, but Python will ***try*** to run it

```
except ZeroDivisionError:
    print("Can't divide by zero!")
```

If the `den` is 0, this code will run

```
except (NameError, ValueError):
    print("Problem with the variables or values!")
```

If there was a problem with a variable name or value, this code will run

```
except:
    print("Some other problem!")
```

For any other exception, this will run

```
print("Moving on!")
```

Unless the program crashes, this code will run

Using more about exceptions

- Exceptions have details
  - You can save and cast the exception to strings to get details
  - You can show the user the problem – and even handle them separately

- The `else` keyword can be used after any exception handling code
  - The code in the `else` block runs only if no exceptions occurred
  - Each are separate blocks though, so be mindful of scoping issues

Example exception handling

```
try:
    num = float(input("Numerator: "))
    den = float(input("Denominator: "))
    result = num/den
except ZeroDivisionError:
    print("Can't divide by zero!")
except (NameError, ValueError):
    print("Problem with the variables or values!")
except:
    print("Some other problem!")
else:
    print(result)
print("Moving on!")
```

```
Numerator: 5
Denominator: 0
Can't divide by zero!
Moving on!

Numerator: bob
Problem with the variables or values!
Moving on!

Numerator: 5
Denominator: 2
2.5
Moving on!
```

Example exception handling

- Lets translate my C++ code to Python…

```
try:
    ofile = open("MyLifesWork.py", "w")
    ifile = open("MyLifesWork.cpp", "r")
except IOError :
    print("Problems opening a file...")
```

- What if another problem occurred?

Capturing a specific exceptions

- Handles multiple errors now

```
try:
    ofile = open("MyLifesWork.py", "w")
    ifile = open("MyLifesWork.cpp", "r")
except IOError :
    print("Problems opening a file...")
except:
    print("Some problem occurred!")
```

- But I want details!

Capturing a specific exception

- Displays the error message now

```
try:
    ofile = open("MyLifesWork.py", "w")
    ifile = open("MyLifesWork.cpp", "r")
except IOError :
    print("Problems opening a file...")
except Exception as e:
    # Exception is the "generic" case
    # Captures all other exceptions
    print("Problems:" + str(e))
```

- Only translate if everything went well…

Capturing a specific exception

- Only translates if there's no problem

```
try:
    ofile = open("MyLifesWork.py", "w")
    ifile = open("MyLifesWork.cpp", "r")
except IOError :
    print("Problems opening a file...")
except Exception as e:
    print("Problems:" + str(e))
else:
    print("We're clear to translate!")
```

- What if there's a problem with only one of the files?
- We may open for writing successfully, but not for reading…
- Now there's a rogue file open

Full example

```
try:
    ofile = open("MyLifesWork.py", "w")
    ifile = open("MyLifesWork.cpp", "r")

except IOError :
    print("Problems opening a file...")
    try:
        ofile.close()
    except:
        print("Couldn't close output file!")
    try:
        ifile.close()
    except:
        print("Couldn't close input file!")

except Exception as e:
    # Exception is the "generic" case
    # Captures all other exceptions
    print("Problems:" + str(e))

else:
    print("We're clear to translate!")

print("All done!")
```

Exceptions summary

1. Run statements in **`try`**

2. If no errors
   a. Skip **`except`** and run **`else`**

3. If a statement in **`try`** raises exception
   a. Skip the remaining statements in **`try`**
   b. Check for any **`except`** block that matches the raised exception
   c. If no matching **`except`** block, exit the program