Introduction to Handling Data Part 1

Week 1: Computer Programming for Data Scientists (7CCSMCMP)

30 Sept 2016

Topics

Data-centric Workflows

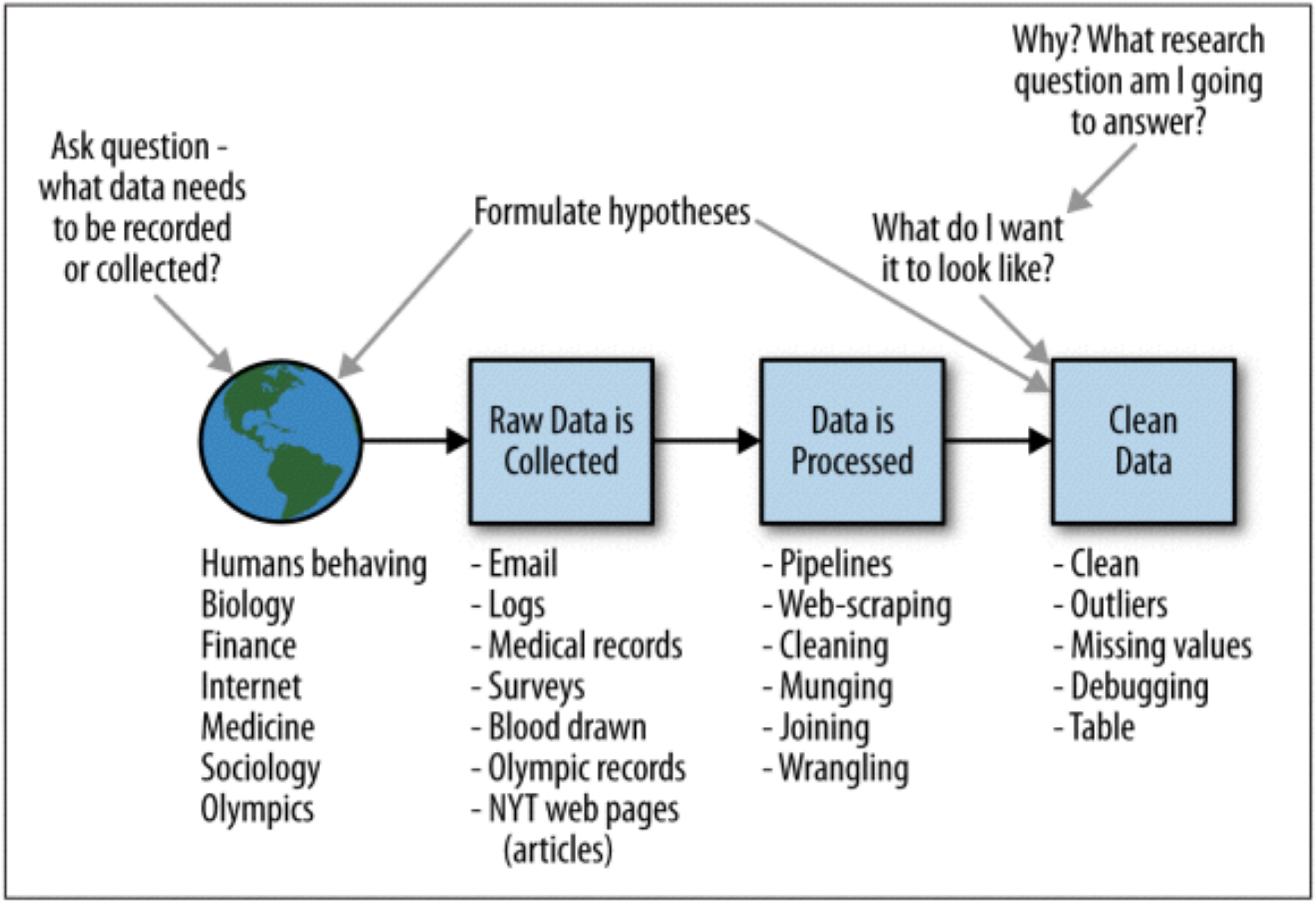
File-Handling

Exception-Handling

String Formatting

CSV Files - reading and writing with `csv`

Data-centric Workflows



From Schutt, R and O'Neil, C (2013). *Doing Data Science*. O'reilly Publishing. pp 43-44

Data-centric Workflows - Stages

1. Data is **extracted** from a data source (a file / database)
   - one or many *heterogeneous* datasets

2. Data is **processed**, and analyzed.
   - usually transformed first into an appropriate structure / format

3. Data is **written** / exported (to a file / database).
   - data format depends on goal / application, i.e. *machine readable* vs. *human readable*, or *both*

Data-centric Workflows - Stages

1. Data is extracted from a data source (a file / database)
   - one or many *heterogeneous* datasets

Extract

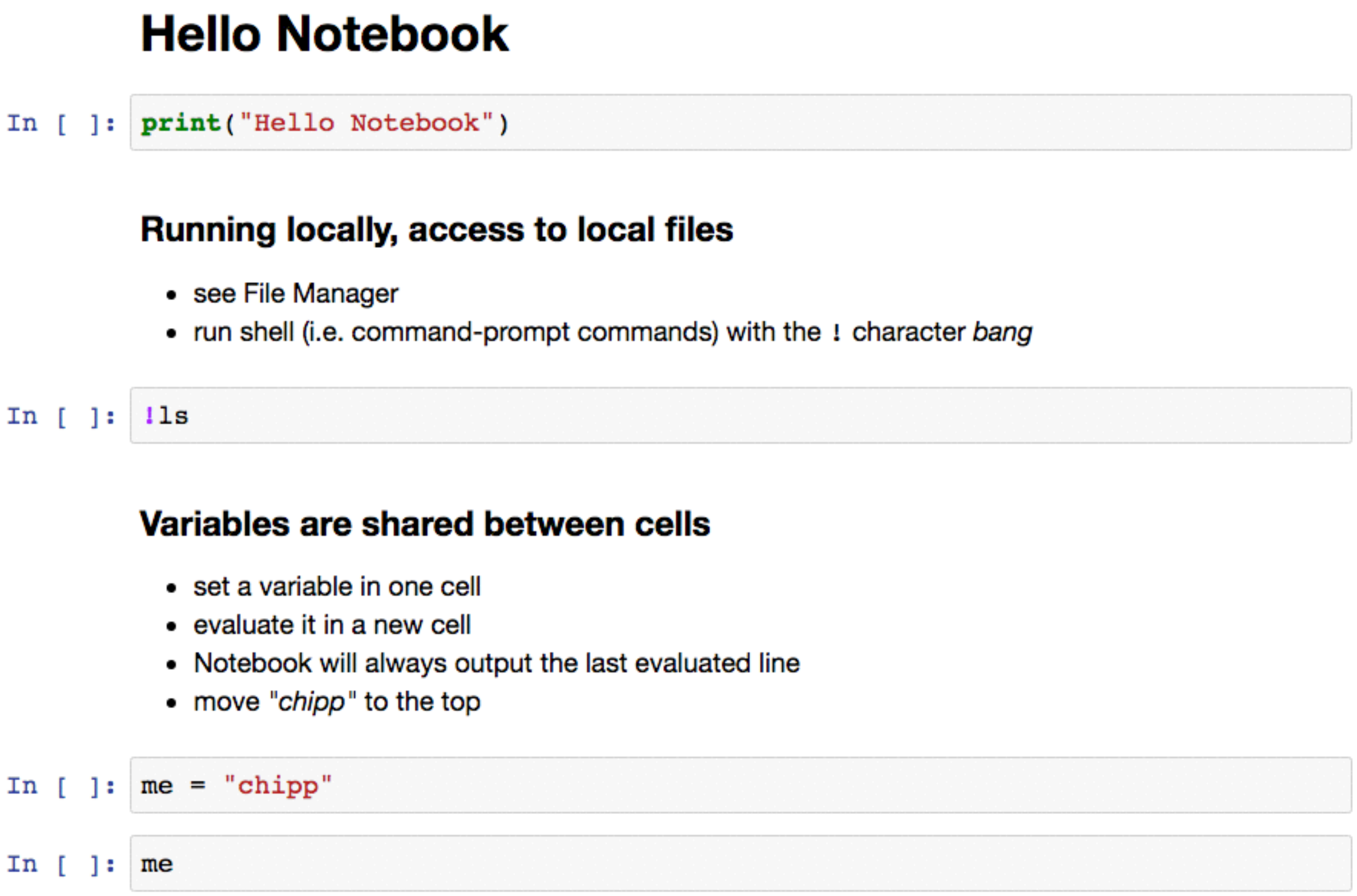
2. Data is processed, and analyzed.
   - usually transformed first into an appropriate structure / format

Transform

3. Data is written / exported (to a file / database).
   - data format depends on goal / application, i.e. *machine readable* vs. *human readable*, or *both*

Load

Analogous to the 3 stages in ETL workflows for databases and data warehouses



Hello Notebook

```
In [ ]: print("Hello Notebook")
```

Running locally, access to local files

- see File Manager
- run shell (i.e. command-prompt commands) with the `!` character *bang*

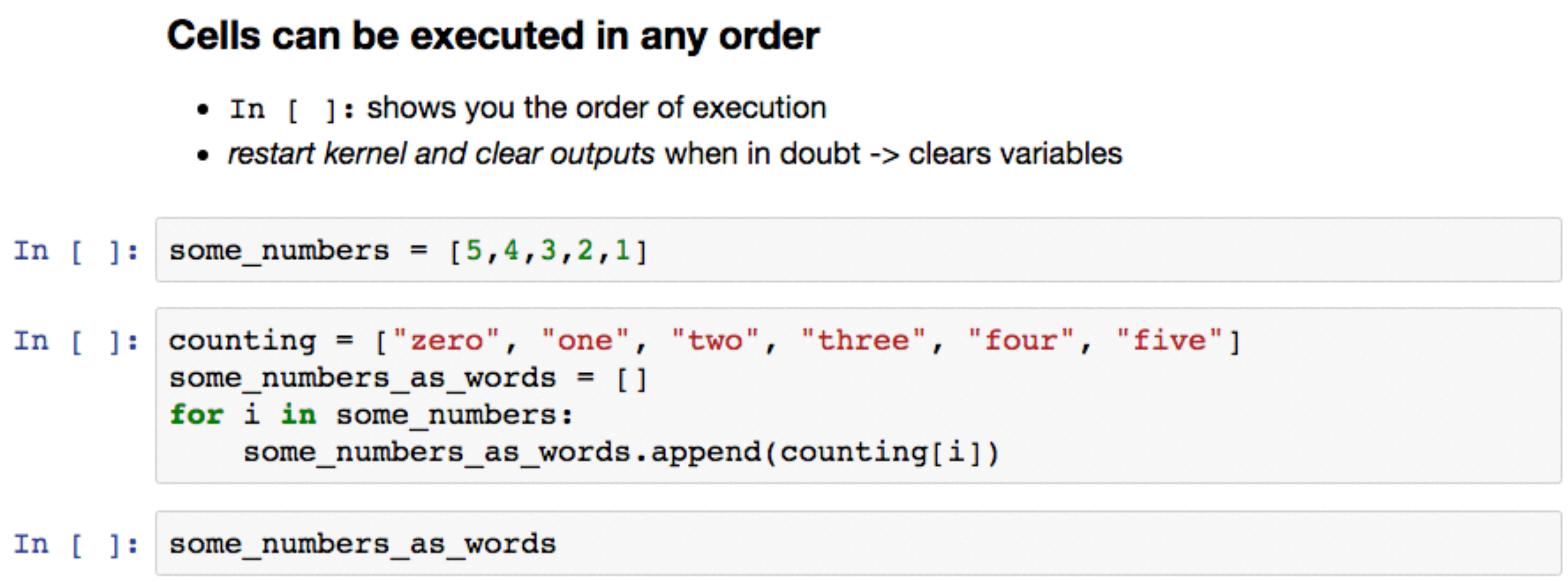
```
In [ ]: !ls
```

Variables are shared between cells

- set a variable in one cell
- evaluate it in a new cell
- Notebook will always output the last evaluated line
- move "*chipp*" to the top

```
In [ ]: me = "chipp"
```

```
In [ ]: me
```



Cells can be executed in any order

- `In [ ]:` shows you the order of execution
- *restart kernel and clear outputs* when in doubt -> clears variables

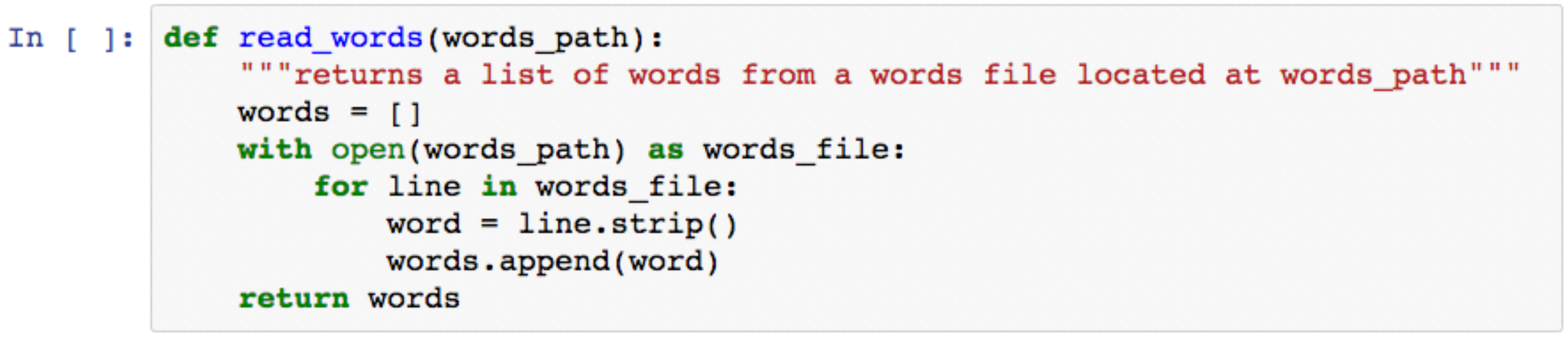
```
In [ ]: some_numbers = [5,4,3,2,1]
```

```
In [ ]: counting = ["zero", "one", "two", "three", "four", "five"]
        some_numbers_as_words = []
        for i in some_numbers:
            some_numbers_as_words.append(counting[i])
```

```
In [ ]: some_numbers_as_words
```

File-Handling

- revisit and dissect this week's lab `read_words` function



```
In [ ]: def read_words(words_path):
            """returns a list of words from a words file located at words_path"""
            words = []
            with open(words_path) as words_file:
                for line in words_file:
                    word = line.strip()
                    words.append(word)
            return words
```

File-Handling - file handles

- use the `open()` function returns a file handle (also known as *file descriptor*)
- two arguements: `filepath` and `mode`
- the `filepath` to the data file is relative to the notebook (just like a python script)
- `modes` are defined as character in a string:

mode	**how it opens the file**
"r"	reading **default**
"w"	writing (erase whatever was in the file before)
"a"	writing by appending to the end of the file (i.e. no erasing)

- when you are done with reading from the file, you should close it by calling the file handle's `close()` function

File-Handling - file handles

- let's take a look at the `words7.txt` file (preview it in the File Manager)
- have `words7.txt` in the **same** directory as your script

In [ ]:

```
words_file = open("words7.txt", "r")
print(words_file.read()) # reads in the file all at once!
words_file.close()
```

File handles can be used in a loop

- it can be used as an *iterator* (more on that in a later lecture)

```
In [ ]: words_file = open("words7.txt", "r")
        for line in words_file: # read in line-by-line
            # print("-------------------------------------")
            print(line)
        words_file.close()
```

Remove the blank space

- notice the extra blank new-lines, we can use the `string.strip()` function to remove this extra blank space

```
In [ ]: words_file = open("words7.txt", "r")
        for line in words_file:
            print(line.strip())
        words_file.close()
```

Store the words in a list

- remember that the `read_words` function would return a list of words

```
In [ ]: words = [] # start a new list

        # open and close the "words7.txt" file
        words_file = open("words7.txt", "r")
        for line in words_file:
            word = line.strip() # clear the newlines
            words.append(word) # add the word to the words list
        words_file.close()

        # show the first 5 words (for easier viewing)
        words[:5]
```

Use a context `with` to open (and close) the file

- annoying to worry about always closing the file

In [ ]:

```
words = [] # start a new list

# open and close the "words7.txt" file
words_file = open("words7.txt", "r")
for line in words_file:
    word = line.strip() # clear the newlines
    words.append(word) # add the word to the words list
words_file.close()

# show the first 5 words (for easier viewing)
words[:5]
```

Use a context `with` to open (and close) the file

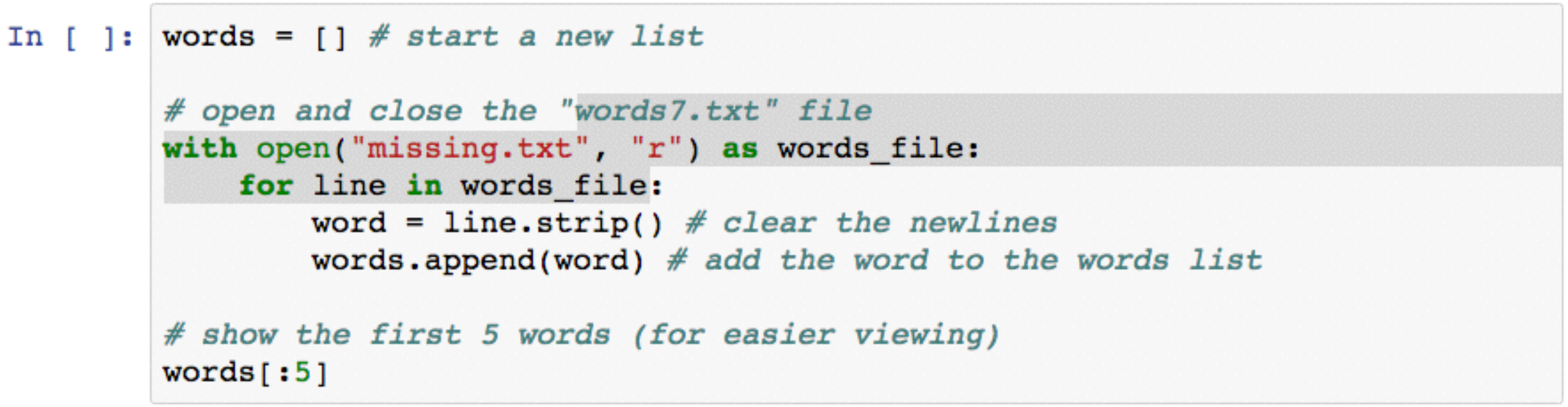
- it's common to use the `with` statement to open the file
- `with` creates a *context*, which is a code block where the file handle will be automatically *closed* at the end of it
- note how the code starts to read almost human "*with open...*"

In [ ]:
```
words = [] # start a new list

# open and close the "words7.txt" file
with open("words7.txt", "r") as words_file:
    for line in words_file:
        word = line.strip() # clear the newlines
        words.append(word) # add the word to the words list

# show the first 5 words (for easier viewing)
words[:5]
```

Exeception Handling

- *exception* handling is a method of managing errors in python
- errors can be *fatal* (programme crashes / code running in cell stops)
- or, errors can be *non-fatal* (a warning is issued and executing continues)

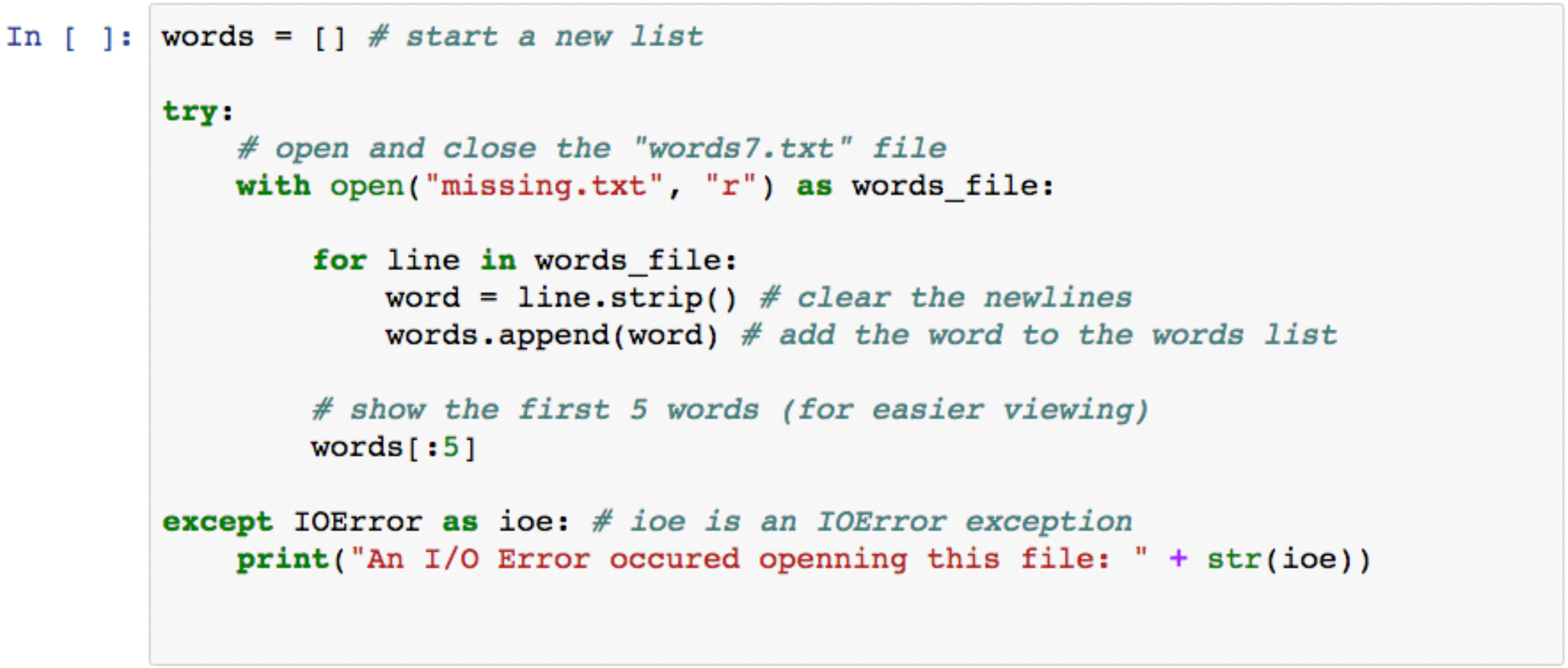
- in jupyter notebook, the exception is printed



```
In [ ]: words = [] # start a new list

        # open and close the "words7.txt" file
        with open("missing.txt", "r") as words_file:
            for line in words_file:
                word = line.strip() # clear the newlines
                words.append(word) # add the word to the words list

        # show the first 5 words (for easier viewing)
        words[:5]
```

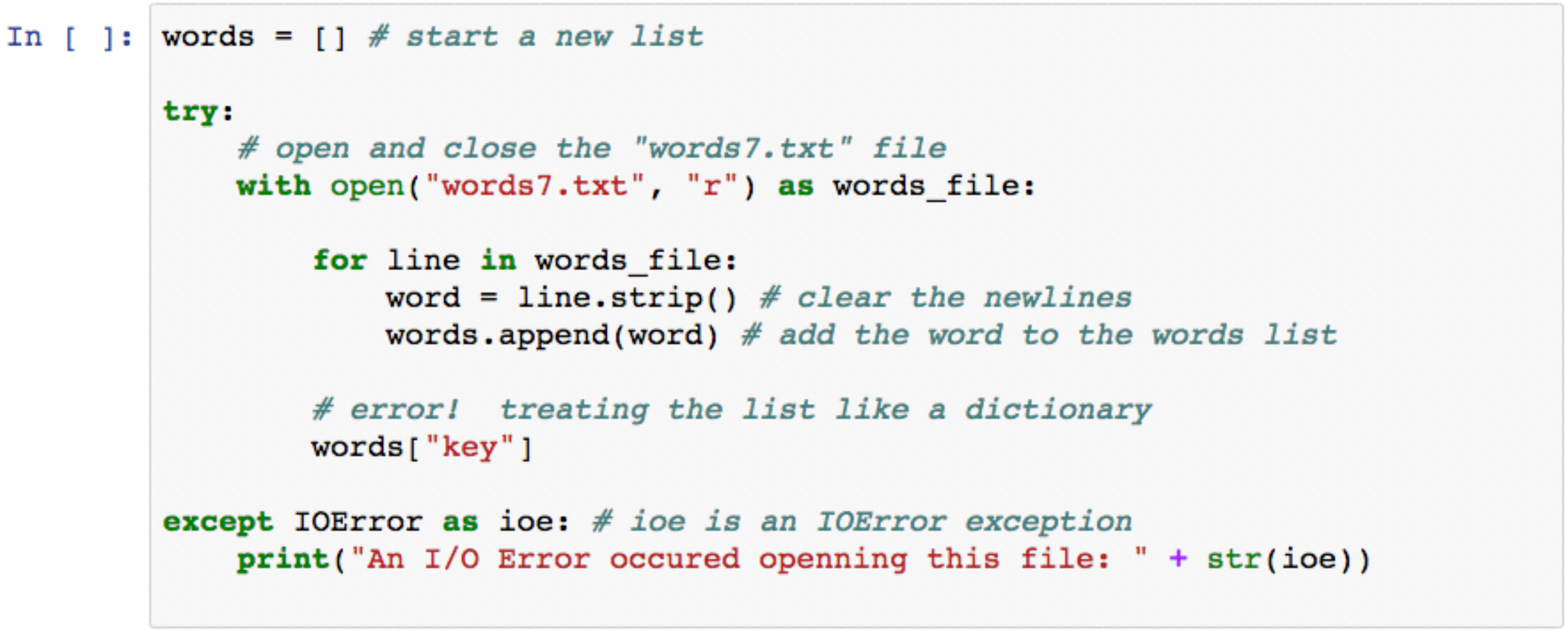
Handling Exceptions

- handle exception with the `try` and `except` statements
- enclose the code you want to *skip* when there is an error



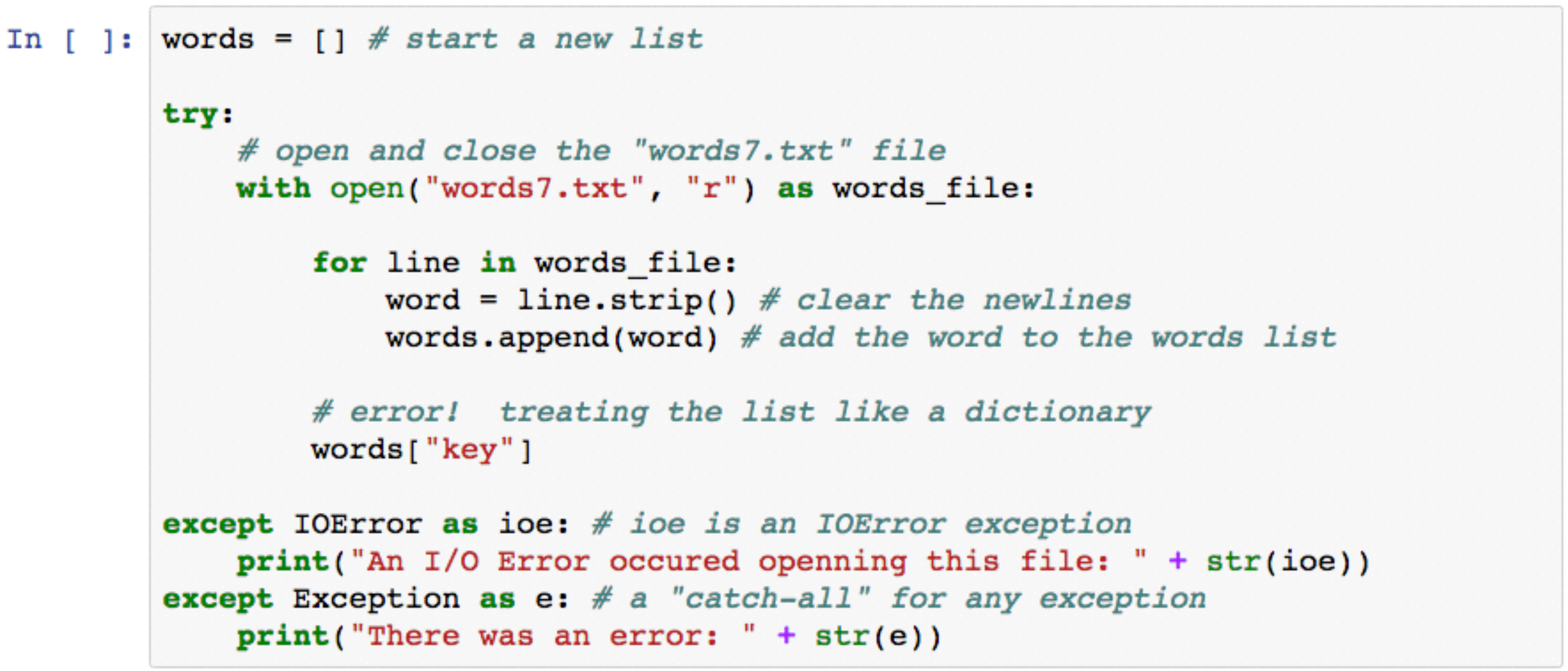
```
In [ ]: words = [] # start a new list

        try:
            # open and close the "words7.txt" file
            with open("missing.txt", "r") as words_file:

                for line in words_file:
                    word = line.strip() # clear the newlines
                    words.append(word) # add the word to the words list

                # show the first 5 words (for easier viewing)
                words[:5]

        except IOError as ioe: # ioe is an IOError exception
            print("An I/O Error occured openning this file: " + str(ioe))
```

Other Exceptions exist (i.e. TypeError)



```
In [ ]: words = [] # start a new list

        try:
            # open and close the "words7.txt" file
            with open("words7.txt", "r") as words_file:

                for line in words_file:
                    word = line.strip() # clear the newlines
                    words.append(word) # add the word to the words list

                # error!  treating the list like a dictionary
                words["key"]

        except IOError as ioe: # ioe is an IOError exception
            print("An I/O Error occured openning this file: " + str(ioe))
```

- can have multiple `except` statements (evaluated in order)
- `except Exception` catches any type of Exception

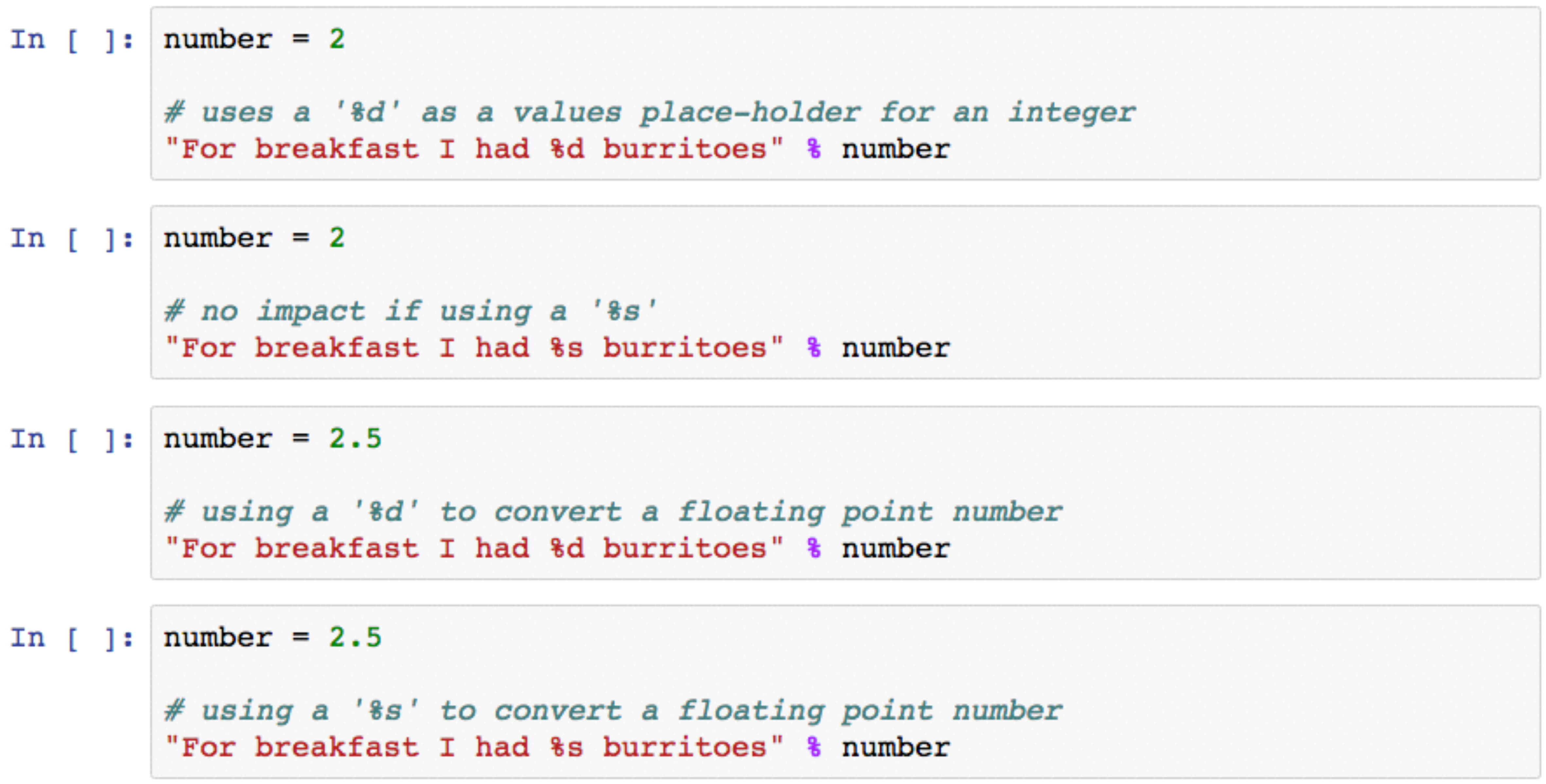


```
In [ ]: words = [] # start a new list

        try:
            # open and close the "words7.txt" file
            with open("words7.txt", "r") as words_file:

                for line in words_file:
                    word = line.strip() # clear the newlines
                    words.append(word) # add the word to the words list

                # error!  treating the list like a dictionary
                words["key"]

        except IOError as ioe: # ioe is an IOError exception
            print("An I/O Error occured openning this file: " + str(ioe))
        except Exception as e: # a "catch-all" for any exception
            print("There was an error: " + str(e))
```

String Formatting

- up until now, we have used *string concatenation* and `str()` to format output
- *string formatting* is more common way of building strings
  - https://docs.python.org/2/library/stdtypes.html#string-formatting
- string formating has a *format* and *values* separated by a `%` (i.e. the interpretor operator)
  - `format % values`
- formats depend on the desired conversion of the values into a string

In [ ]:
```
breakfast = "burrito"

# uses a '%s' as a values place-holder for a string
"For breakfast I had a %s" % breakfast
```



```
In [ ]: number = 2

        # uses a '%d' as a values place-holder for an integer
        "For breakfast I had %d burritoes" % number
```

```
In [ ]: number = 2

        # no impact if using a '%s'
        "For breakfast I had %s burritoes" % number
```

```
In [ ]: number = 2.5

        # using a '%d' to convert a floating point number
        "For breakfast I had %d burritoes" % number
```

```
In [ ]: number = 2.5

        # using a '%s' to convert a floating point number
        "For breakfast I had %s burritoes" % number
```

Use a *tuple* to format more than 1 value

In [ ]:
```
meal = "breakfast"
number = 0.1

# using a '%s' to convert a floating point number
"For %s I had %s burritoes" % (meal, number)
```

Formatted string are just strings (can be used in `print()` functions for example)

In [ ]:
```
meal = "dinner"
number = 4
food = "pizzas"

print("For %s I had %d %s" % (meal, number, food))
```

File Handling - write a file

- use the `write()` function on the file descriptor

In [ ]:

```python
# write my meal to a file
meal = "lunch"
number = 2
food = "carrots"

try:
    # open and close the "meal_plan.txt" file write-only
    with open("meal_plan.txt", "w") as meal_file:
        meal_file.write("For %s I had %d %s" % (meal, number, food))

except IOError as ioe: # ioe is an IOError exception
    # NOTE - the exception is converted to its string representation
    print("An I/O Error occured openning this file: %s" % ioe)
```

File Handling - write a file - many strings

- use a for-loop to write many strings to a file

In [ ]:

```
# my meals
meals =[("breakfast", 2, "muffins"), ("lunch", 0, "carrots"),
        ("dinner", 3, "apples")]

try:
    # open and close the "meal_plan.txt" file write-only
    with open("meal_plan.txt", "w") as meal_file:
        for m in meals:
            print(m) # print the tuple to output
            meal_file.write("For %s I had %d %s" % m)

except IOError as ioe: # ioe is an IOError exception
    # NOTE - the exception is converted to its string representation
    print("An I/O Error occured openning this file: %s" % ioe)
```

File Handling - write a file - using newline

- need to add a *newline* character `'\n'` to the string when printing it
- the `'\'` and the `'n'` together as `'\n'` count as *one character* when written

In [ ]:

```
# my meals
meals =[("breakfast", 2, "muffins"), ("lunch", 0, "carrots"),
        ("dinner", 3, "apples")]

try:
    # open and close the "meal_plan.txt" file write-only
    with open("meal_plan.txt", "w") as meal_file:
        for m in meals:
            print(m) # print the tuple to output
            meal_file.write("For %s I had %d %s\n" % m) # add newline

except IOError as ioe: # ioe is an IOError exception
    # NOTE - the exception is converted to its string representation
    print("An I/O Error occured openning this file: %s" % ioe)
```

File Handling - write a file - randomise strings

- let python decide what I eat for today's meals (random food)
- pick food using the `random.choice()` function: https://docs.python.org/2/library/random.html#random.choice
- pick amount of food using the 'random.randin()' function: https://docs.python.org/2/library/random.html#random.randint

```
In [ ]: import random

        meals =["breakfast", "lunch", "snack", "dinner"]
        foods = ["muffins", "carrots", "apples", "Yorkshore puddings",
                 "pieces of sushi", "burritoes"]

        try:
            # open and close the "meal_plan.txt" file write-only
            with open("meal_plan.txt", "w") as meal_file:
                for m in meals:
                    food = random.choice(foods)
                    amount = random.randint(0,5) # represents 1 to 5 inclusive
                    meal_file.write("For %s I had %d %s\n" % (m, amount, food))

        except IOError as ioe: # ioe is an IOError exception
            # NOTE - the exception is converted to its string representation
            print("An I/O Error occured openning this file: %s" % ioe)
```

CSV Files

- CSV files are Comma-Separated Value files
- most common *table* data format
  - every CSV file as a table
  - every line is a *row*
  - lines are values separated by commas (i.e. into *columns*)
- many **dialects** for CSV files
- *example* - Parents' country of birth from Office of National Statistics (preview in file manager)
- Python has a built-in csv module (so you don't have to write your own parsers)
  - https://docs.python.org/2/library/csv.html

In [ ]:
```
import csv # import the csv module

try:
    with open("data/mothers.csv", "r") as mothers_fd: # open a file context
        csv_data = csv.reader(mothers_fd)
        mothers = list(csv_data) # converts the *iterator* into a list
except IOError as ioe:
    print("IOError: " + str(ioe))
```

In [ ]:
```
# look at the first 5 elements in the list - what are they?
print(mothers[:5])
```

```
[
  ['', '', '', '', 'Mothers born outside United Kingdom', '', '', '', '', '', '',
   '', '', ''],
  ['', '', '', '','', '', 'European Union', '', '', '',  '', '', '', ''],
  ['Code', 'Area', 'All Live Births','Mothers Born within United Kingdom',
   'Total', 'Percentage of live births to non-UK born mothers', 'Total',
   'New EU', 'Rest of Europe (non EU)',
   'Middle East and Asia', 'Africa', 'Rest of World ', '', ''],
   ['', '', '', '', '', '', '', '', '', '', '', '', '', ''],
   ['E09000002', 'Barking and Dagenham', '3,796', '1,411', '2,383', '62.8',
    '595', '527', '123', '692', '915','58', '', ''] , ...
]
```

- the `csv.reader()` pulls the data in as a *list of lists*
- order is **row first** and then columns
- can use two `[ ]` to reference a single value

```
In [ ]: print(mothers[0][4])
```

```
In [ ]: print(mothers[4][1])
```

Parsing with csv.DictReader()

- CSV files have the notion of *headers*, or named columns
- rather than use list indices to reference a column, it would be nice to use the column names as strings
- would like to use a *list of dictionaries* rather than a *list of lists*

```
In [ ]: print(mothers[4]["Area"])
```

Parsing with csv.DictReader()

- use `csv.DictReader()` instead of `csv.reader()` to parse the data into a `dict` instead of a `list`
- need to define the column names, or *fields* (i.e. "keys") for the `dict` as an arguement to `csv.DictReader()`

```
In [ ]: import csv # import the csv module

        try:
            with open("data/mothers.csv", "r") as mothers_fd: # open a file context
                csv_data = csv.DictReader(mothers_fd, fieldnames=['Code','Area'])
                mothers = list(csv_data) # converts the *iterator* into a list
        except IOError as ioe:
            print("IOError: " + str(ioe))
```

```
In [ ]: print(mothers[4]["Area"]) # Area and it's Code
        print(mothers[4]["Code"])
```

```
In [ ]: mothers[4] # here's what it looks like as a dict
```

Cleaning raw data

- notice many empty rows (i.e. records) in this raw data
- need to clean it by removing the empty records
- identify an empty record by whether it is missing the "Code" field, if so, remove it
- whether or not you decide to remove records and rows depends on the dataset and the task at hand (i.e. you may want missing values)

```
In [ ]: # using the mothers list of dictionaries from previous example

        # show the number of all records
        print "Number of records: %d" % len(mothers)

        # show *all* of the records
        for rec in mothers:
            print "%s %s" % (rec["Code"], rec["Area"])
```

Cleaning raw data - removing records

- Use the list `remove()` function to remove the record

In [ ]:
```
for rec in mothers:
    if len(rec['Code']) == 0: # test the length of the field value
        mothers.remove(rec)

# show the number of remaining "cleaned" records
print "Number of clean records: %d" % len(mothers)

# print out the "Code" and "Area, again
for rec in mothers:
    print "%s %s" % (rec["Code"], rec["Area"])
```

Writing out CSV files

- use the `csv.DictWriter` to write-back the cleaned dataset
- need to define the `fieldnames` again
- use `writeheader()` write the header for the CSV file
- `writerows` outputs all of the records

```
In [ ]: import csv

        with open('mothers_areas.csv', 'w') as csvfile:
            # define fieldnames again
            writer = csv.DictWriter(csvfile, fieldnames=['Code', 'Area'],
                                    extrasaction='ignore')

            writer.writeheader() # writes the CSV column header
            writer.writerows(mothers) # all of the records,
                                      # ignore any non-specified columns

        # look at mothers_areas.csv in File Manager
```

Summary

- Reviewed Jupyter Notebook and its interaction
- File-Handing using `with`: reading / writing
- String Formating using the `%` operator
- CSV Files and using the `csv` module
    - reading into list-of-lists with `csv.reader`
    - reading into a list-of-dictionaries with `csv.DictReader`
    - cleaning data
    - writing a list-of-dictionaries with `csv.DictWriter`