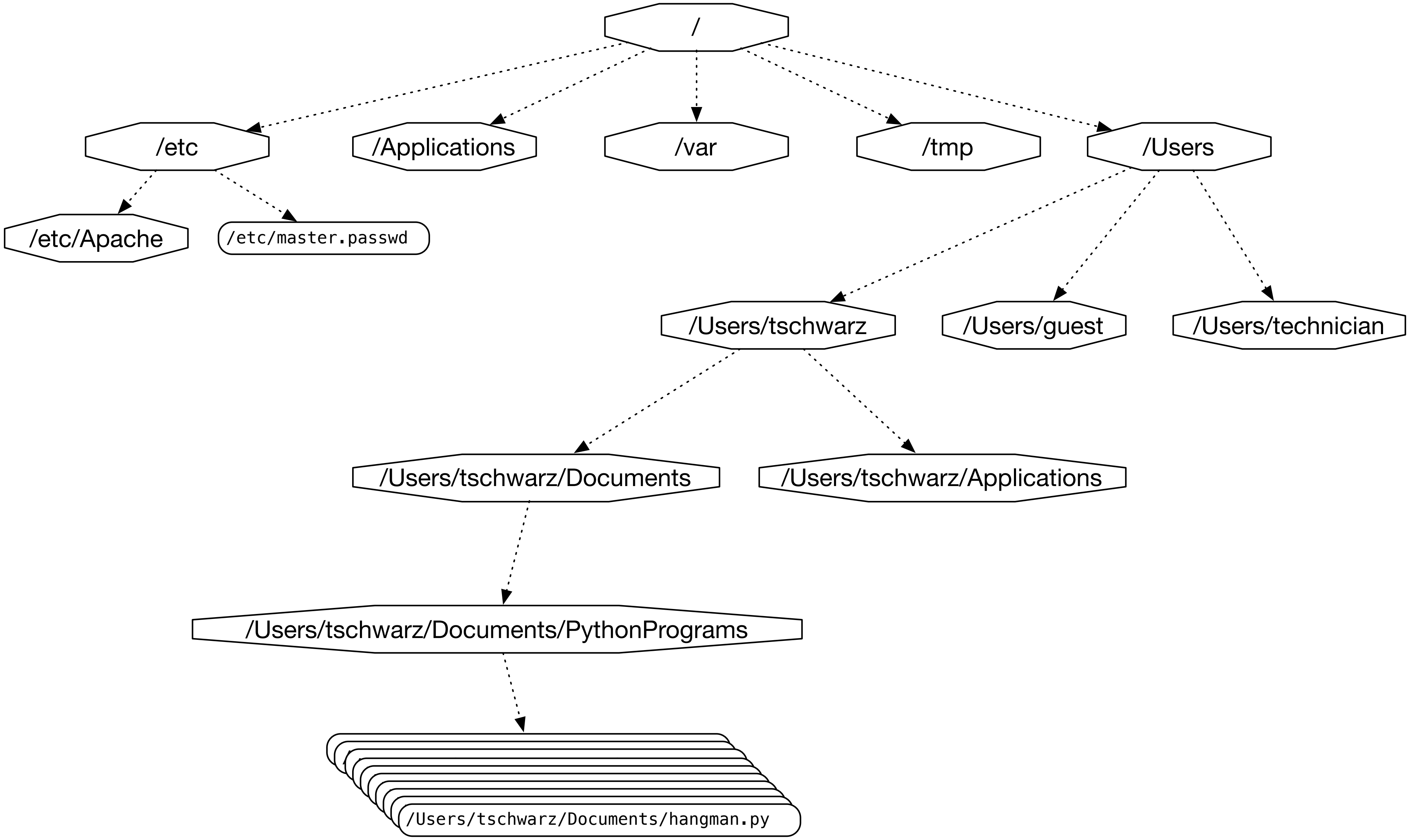
Dealing with Files

Thomas Schwarz, SJ

Files

- Files
  - Basic container of data in modern computing system
  - Organized into a hierarchy of directories

Files


/
/etc
/Applications
/var
/tmp
/Users
/etc/Apache
/etc/master.passwd
/Users/tschwarz
/Users/guest
/Users/technician
/Users/tschwarz/Documents
/Users/tschwarz/Applications
/Users/tschwarz/Documents/PythonPrograms
/Users/tschwarz/Documents/hangman.py

A small subset of directories a

Files in Python

- Files accessed in
  - text mode
    - Contents interpreted according to encoding
  - binary mode
    - Contents not interpreted

Files in Python

- Python interacts by files through
  - reading
  - writing / appending
  - both

Files in Python

- Files need to be opened
  - File given by name
    - Relative path: Navigation from directory of the file
    - Absolute path: Navigation from the root of the file system

Files in Python

- File Name Examples:
    - Absolute path on a Mac / Unix

```
/Users/tjschwarzsj/Google Drive/AATeaching/Python/Programs/pr.py
```

- Relate path on a Mac / Unix
    - "../" means move up on directory

```
pr.py

../Slides/week7.key
```

Files in Python

- Windows uses backward slashes to separate directories in a file name
    - Sometimes need to be escaped: \\
    - Absolute paths need to include drive name:
        - c:\\users\\tschwarz\\My Documents\\Teaching\\temp.py
- *We will typically read and create files in the same directory as the python program is located*

Files in Python

- Before files are used, program needs to open them
- After they are being used, program should close them
  - Will automatically closed when program terminates
  - Long-running programs could hog resources

Opening Files in Python

- File objects have normal variable names

```
inFile = open(“data.txt”,”w”)
```

- opens a file “data.txt” in write mode

- open takes :
  - file name — absolute / relative path
  - mode — r (read), w (write), a (appending)
  - mode — b (binary), “” or t (text mode)

Closing Files in Python

- We close file by invoking close
  - `inFile.close()`

Why we need to close files

- Files are automatically closed when the program terminates
- When one application has opened a file for writing it acquires a write lock on the file and no other application can access the file.
- When one application has opened a file for reading, it acquires a read lock on the file and no other application can write to it.
- If you write programs that last more than a few seconds, you do not want to hog files when you do not need them.

With-clauses

- Python 3 allows us to open and close files in a single block (context)

```
with open("twoft8.11.txt") as inFile, open("twoftres8.11.txt",
"w") as outFile:

        #Here you work with the file
```

Processing Files in Python

- We write strings to the file

```
with open(‘somefile.txt’,’wt’) as f:
 f.write(str(500)+”\n")
```

- Redirect print

```
with open(‘somefile.txt’,’wt’) as f:
 print(500, file = f)
```

Processing Files in Python

- Reading files
  - The read-instruction

    ```
    string = inFile.read(10)
    ```

    reads ten bytes of the file
  - Read the entire file

    ```
    with open('somefile.txt', 'rt') as f:
        data = f.read()
    ```

Processing Files in Python

- Reading files
  - Read line by line

```
with open('somefile.txt', 'rt') as f:
    for line in f:
         #process line
```

More String Processing

- To process read lines:
  - `strip()` and its variants `lstrip(), rstrip()`
    - Remove white spaces (default) or list of characters from the beginning & end of the string
  - `split()` creates a list of words separated by white space (default)

```
"This is a sentence with many words in
it.".split()
```

```
['This', 'is', 'a', 'sentence', 'with',
'many', 'words', 'in', 'it.']
```

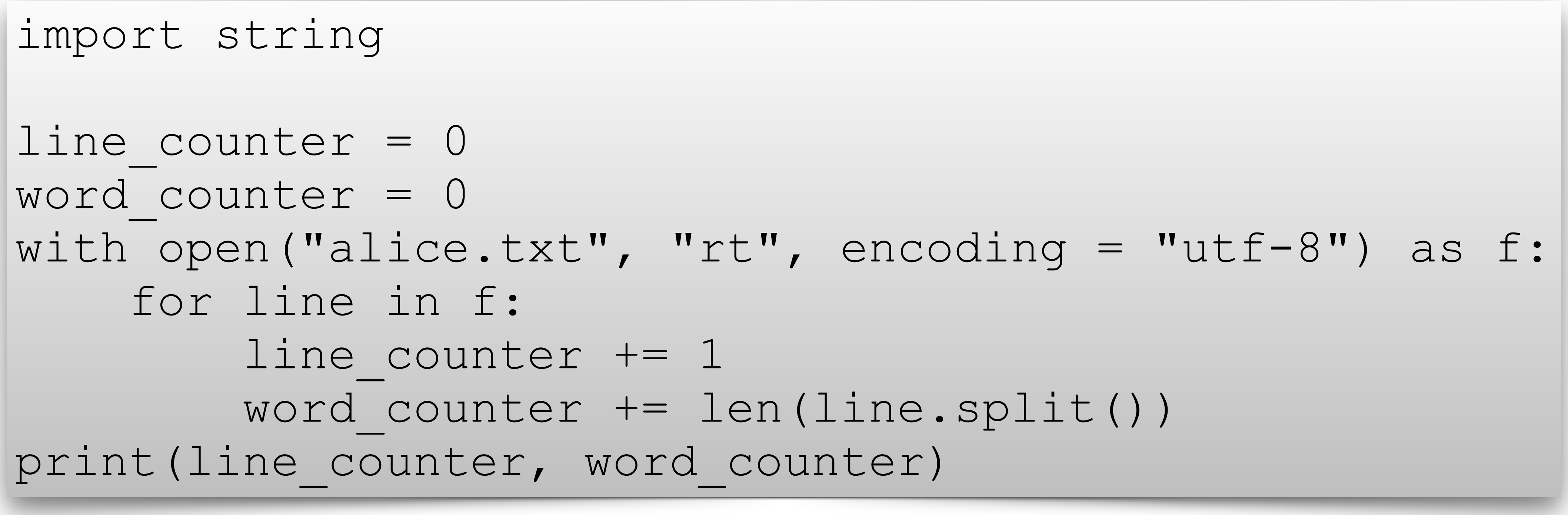
Examples

- Finding all words over 13 letters long in “Alice in Wonderland”
    - Download from Project Gutenberg

```
import string

with open("alice.txt", "rt", encoding = "utf-8") as f:
    for line in f:
        for word in line.split():
            if len(word) > 13:
                print(word)
```

Examples

- Count the number of words and of lines in “Alice in Wonderland”
  - Read the file line by line
    - The number of words in a line is the length of line.split.



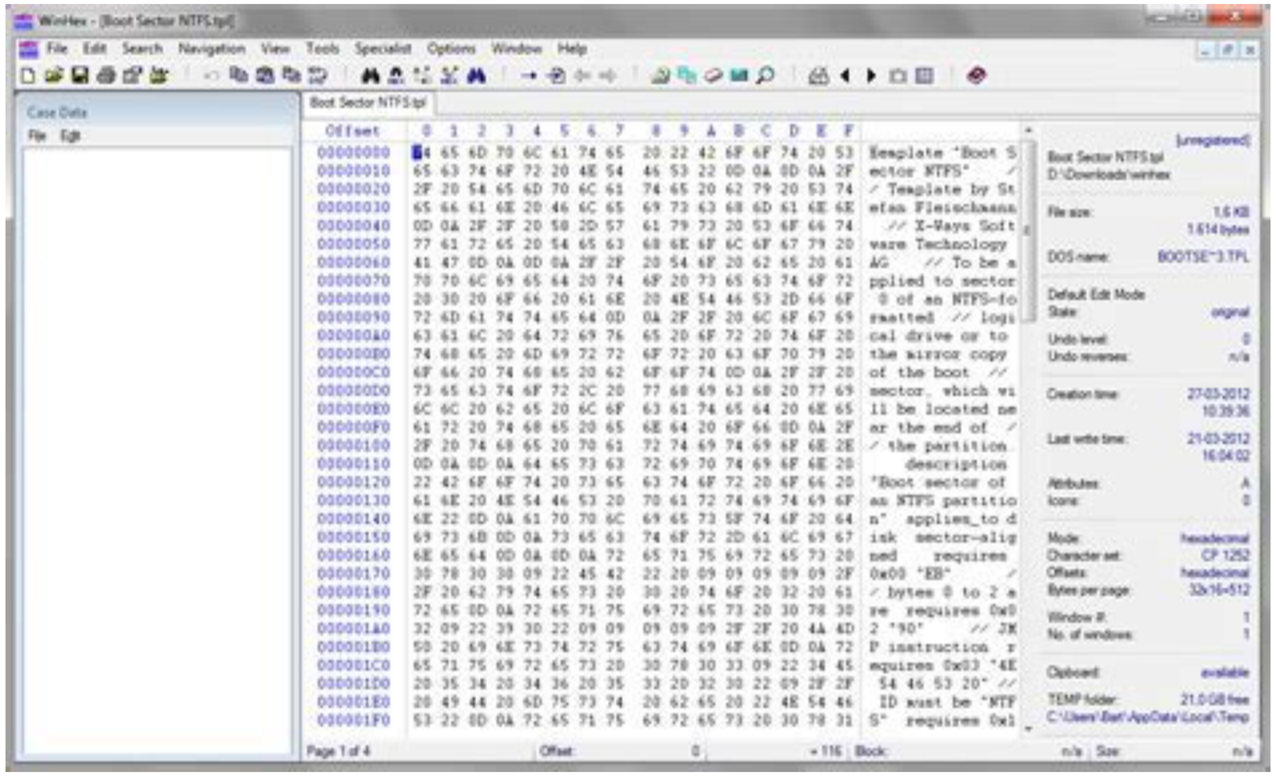
```
import string

line_counter = 0
word_counter = 0
with open("alice.txt", "rt", encoding = "utf-8") as f:
    for line in f:
        line_counter += 1
        word_counter += len(line.split())
print(line_counter, word_counter)
```

Problems with Line Endings

- ASCII code was developed when computers wrote to teleprinters.
    - A new line consisted of a carriage return followed or preceded by a line-feed.
- UNIX and windows choose to different encodings
    - Unix has just the newline character “\n”
    - Windows has the carriage return: “\r\n”
- By default, Python operates in “universal newline mode”
    - All common newline combinations are understood
    - Python writes new lines with the system default
- You could disable this mechanism by opening a file with the universal newline mode disabled by saying:
    - `open(“filename.txt”, newline='')`

Encodings

- Information technology has developed a large number of ways of storing particular data
  - Here is some background



```
 Offset    0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
00000000  54 65 6D 70 6C 61 74 65  20 22 42 6F 6F 74 20 53  Template "Boot S
00000010  65 63 74 6F 72 20 4E 54  46 53 22 0D 0A 0D 0A 2F  ector NTFS"    /
00000020  2F 20 54 65 6D 70 6C 61  74 65 20 62 79 20 53 74  / Template by St
00000030  65 66 61 6E 20 46 6C 65  69 73 63 68 6D 61 6E 6E  efan Fleischmann
00000040  0D 0A 2F 2F 20 58 2D 57  61 79 73 20 53 6F 66 74    // X-Ways Soft
00000050  77 61 72 65 20 54 65 63  68 6E 6F 6C 6F 67 79 20  ware Technology
00000060  41 47 0D 0A 0D 0A 2F 2F  20 54 6F 20 62 65 20 61  AG    // To be a
00000070  70 70 6C 69 65 64 20 74  6F 20 73 65 63 74 6F 72  pplied to sector
00000080  20 30 20 6F 66 20 61 6E  20 4E 54 46 53 2D 66 6F   0 of an NTFS-fo
00000090  72 6D 61 74 74 65 64 0D  0A 2F 2F 20 6C 6F 67 69  rmatted  // logi
000000A0  63 61 6C 20 64 72 69 76  65 20 6F 72 20 74 6F 20  cal drive or to
000000B0  74 68 65 20 6D 69 72 72  6F 72 20 63 6F 70 79 20  the mirror copy
000000C0  6F 66 20 74 68 65 20 62  6F 6F 74 0D 0A 2F 2F 20  of the boot  //
000000D0  73 65 63 74 6F 72 2C 20  77 68 69 63 68 20 77 69  sector, which wi
000000E0  6C 6C 20 62 65 20 6C 6F  63 61 74 65 64 20 6E 65  ll be located ne
000000F0  61 72 20 74 68 65 20 65  6E 64 20 6F 66 0D 0A 2F  ar the end of  /
00000100  2F 20 74 68 65 20 70 61  72 74 69 74 69 6F 6E 2E  / the partition.
00000110  0D 0A 0D 0A 64 65 73 63  72 69 70 74 69 6F 6E 20      description
00000120  22 42 6F 6F 74 20 73 65  63 74 6F 72 20 6F 66 20  "Boot sector of
00000130  61 6E 20 4E 54 46 53 20  70 61 72 74 69 74 69 6F  an NTFS partitio
00000140  6E 22 0D 0A 61 70 70 6C  69 65 73 5F 74 6F 20 64  n"  applies_to d
00000150  69 73 6B 0D 0A 73 65 63  74 6F 72 2D 61 6C 69 67  isk  sector-alig
00000160  6E 65 64 0D 0A 0D 0A 72  65 71 75 69 72 65 73 20  ned    requires
00000170  30 78 30 30 09 22 45 42  22 20 09 09 09 09 09 2F  0x00 "EB"      /
00000180  2F 20 62 79 74 65 73 20  30 20 74 6F 20 32 20 61  / bytes 0 to 2 a
00000190  72 65 0D 0A 72 65 71 75  69 72 65 73 20 30 78 30  re  requires 0x0
000001A0  32 09 22 39 30 22 09 09  09 09 09 2F 2F 20 4A 4D  2 "90"     // JM
000001B0  50 20 69 6E 73 74 72 75  63 74 69 6F 6E 0D 0A 72  P instruction  r
000001C0  65 71 75 69 72 65 73 20  30 78 30 33 09 22 34 45  equires 0x03 "4E
000001D0  20 35 34 20 34 36 20 35  33 20 32 30 22 09 2F 2F   54 46 53 20" //
000001E0  20 49 44 20 6D 75 73 74  20 62 65 20 22 4E 54 46   ID must be "NTF
000001F0  53 22 0D 0A 72 65 71 75  69 72 65 73 20 30 78 31  S"  requires 0x1
```

[unregistered]
Boot Sector NTFS.tpl
D:\Downloads\winhex

File size: 1.6 KB
1,614 bytes
DOS name: BOOTSE~3.TPL

Default Edit Mode
State: original
Undo level: 0
Undo reverses: n/a

Creation time: 27-03-2012 10:39:36
Last write time: 21-03-2012 16:04:02

Attributes: A
Icons: 0

Mode: hexadecimal
Character set: CP 1252
Offsets: hexadecimal
Bytes per page: 32x16=512
Window #: 1
No. of windows: 1

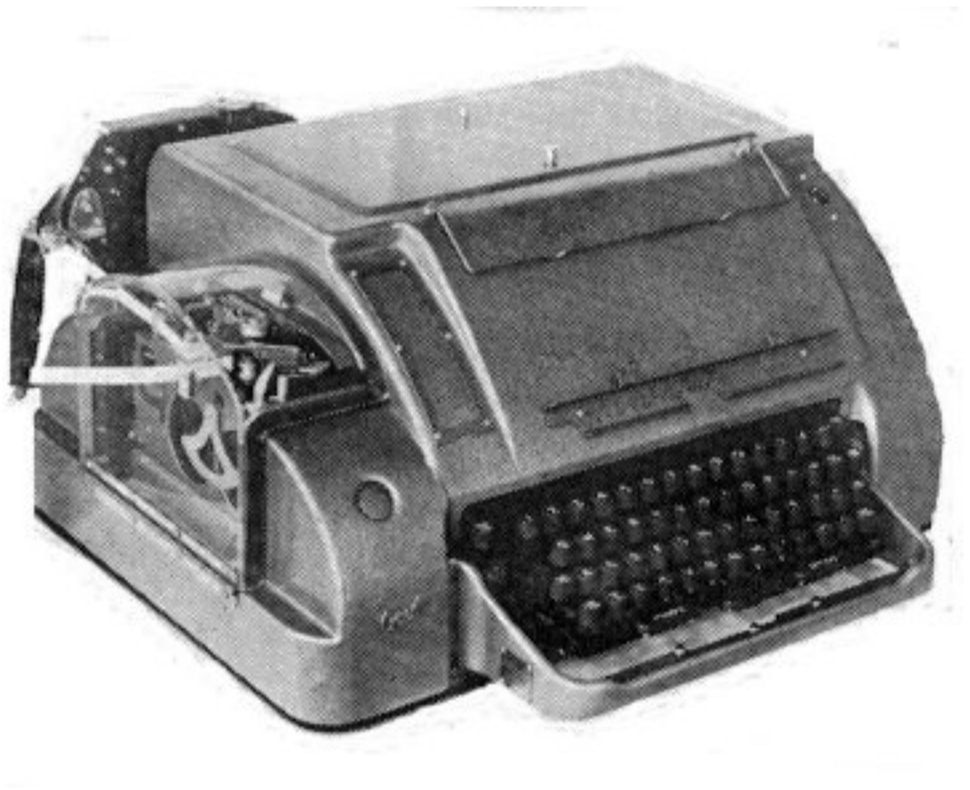
Clipboard: available
TEMP folder: 21.0 GB free
C:\Users\Bart\AppData\Local\Temp

Page 1 of 4 | Offset: 0 | = 116 | Block: n/a | Size: n/a

```
Using a forensics tool (Winhex) in order
to reveal the bytes actually stored
```

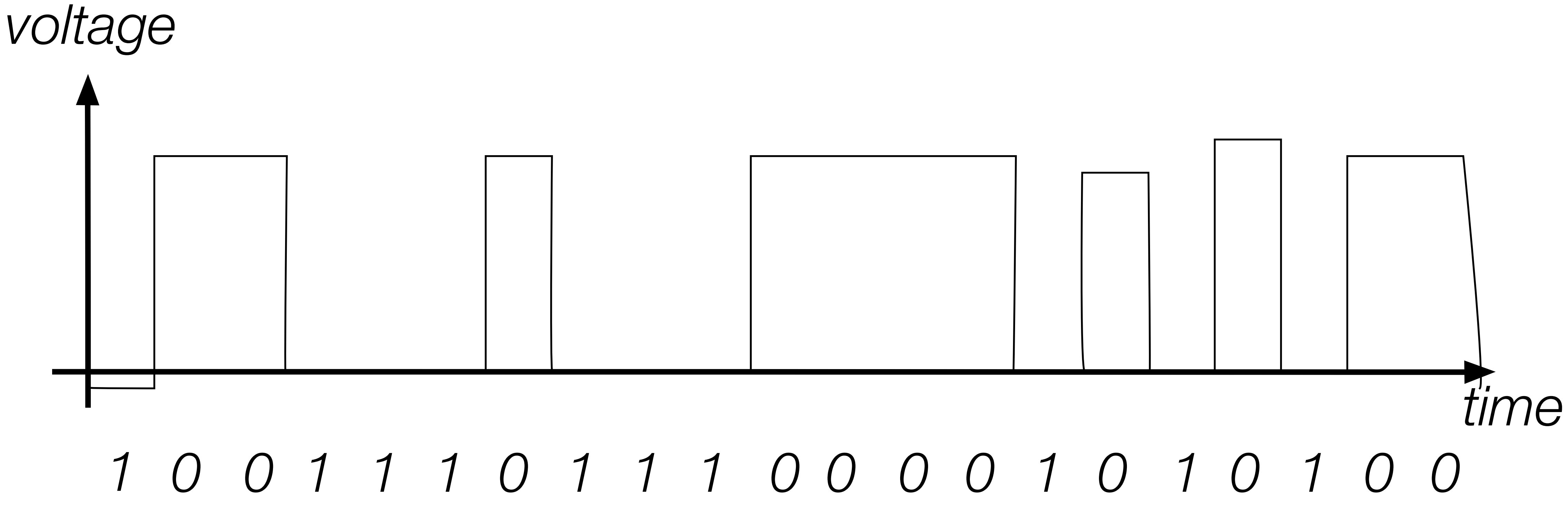
Encodings

- Teleprinters
  - Used to send printed messages
    - Can be done through a single line
    - Use timing to synchronize up and down values



Encodings

- Serial connection:
  - Voltage level during an interval indicates a bit
  - Digital means that changes in voltage level can be tolerated without information loss

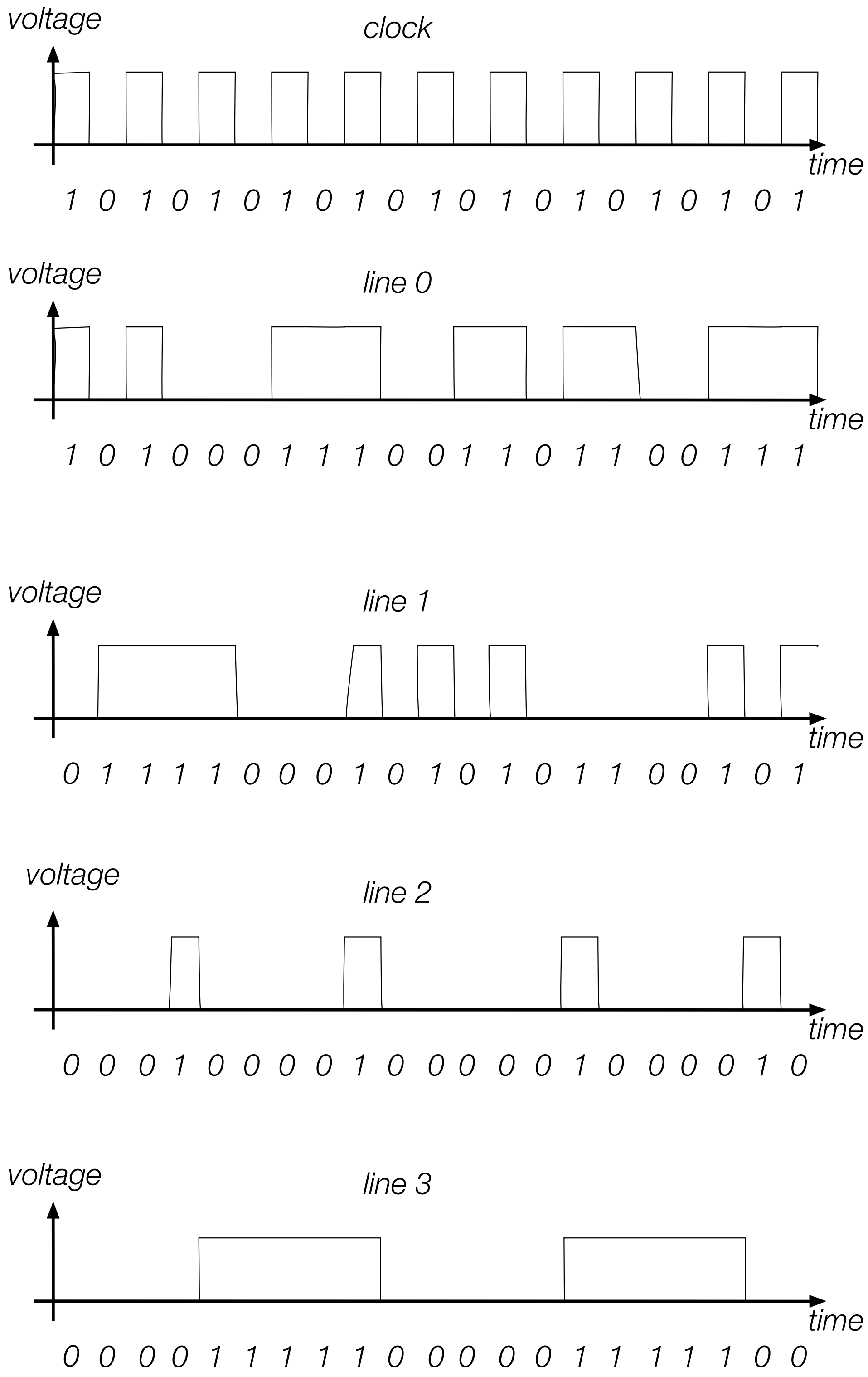


Encodings

- Parallel Connection
  - Can send more than one bit at a time
  - Sometimes, one line sends a timing signal

Encodings

- Sending
  - 1000
  - 0100
  - 1100
  - 0100
  - …
- Small errors in timing and voltage are repaired automatically



Encodings

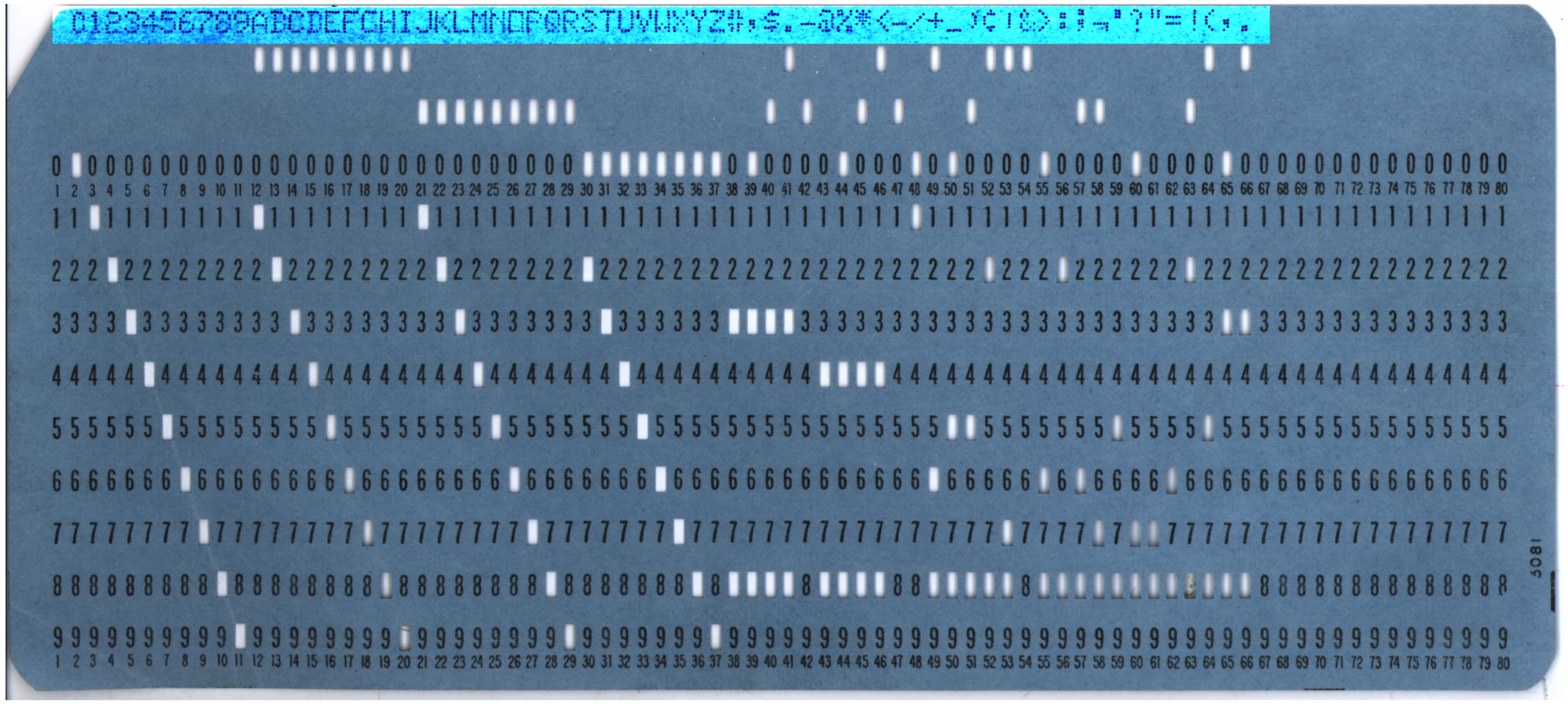
- Need a code to transmit letters and control signals
- Émile Baudot’s code 1870
  - 5 bit code
    - Machine had 5 keys, two for the left and three for the right hand
    - Encodes capital letters plus NULL and DEL
    - Operators had to keep a rhythm to be understood on the other side

Encodings

- Many successors to Baudot’s code
    - Murray’s code (1901) for keyboard
        - Introduced control characters such as Carriage Return (CR) and Line Feed (LF)
        - Used by Western Union until 1950

Encodings

- Computers and punch cards
    - Needed an encoding for strings
        - EBCDIC — 1963 for punch cards by IBM
        - 8b code


0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ#,$.-@%*<-/+_)¢|&>:;¬'?"=!(,.
5081

Encodings

- ASCII — American Standard Code for Information Interchange — 1963
  - 8b code
    - Developed by American Standard Association, which became American National Standards Institute (ANSI)
    - 32 control characters
    - 91 alphanumerical and symbol characters
    - Used only 7b to encode them to allow local variants
  - Extended ASCII
    - Uses full 8b
      - Chooses letters for Western languages

Encodings

- Unicode - 1991
  - “Universal code” capable of implementing text in all relevant languages
  - 32b-code
  - For compression, uses “language planes”

Encodings

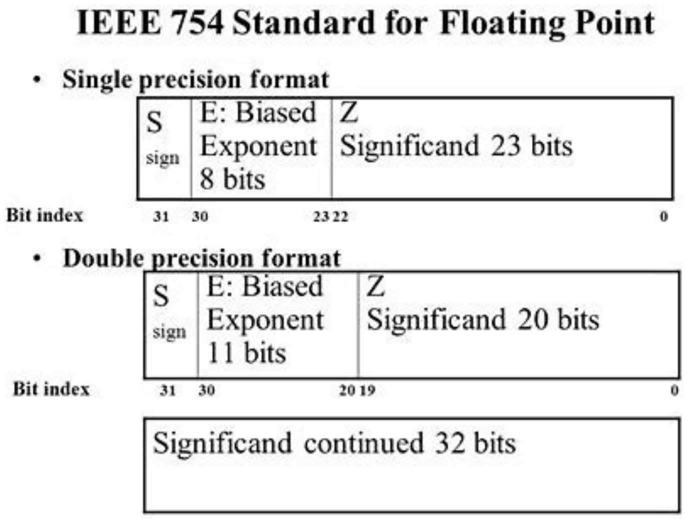
- UTF-7 — 1998
  - 7b-code
    - Invented to send email more efficiently
    - Compatible with basic ASCII
    - Not used because of awkwardness in translating 7b pieces in 8b computer architecture

Encodings

- UTF-8 — Unicode
  - Code that uses
    - 8b for the first 128 characters (basically ASCII)
    - 16b for the next 1920 characters
      - Latin alphabets, Cyrillic, Coptic, Armenian, Hebrew, Arabic, Syriac, Thaana, N’Ko
    - 24b for
      - Chinese, Japanese, Koreans
    - 32b for
      - Everything else

Encodings

- Numbers
  - There is a variety of ways of storing numbers (integers)
    - All based on the binary format
  - For floating point numbers, the exact format has a large influence on the accuracy of calculations
    - All computers use the IEEE standard



IEEE 754 Standard for Floating Point

- Single precision format
- Double precision format

Python and Encodings

- Python “understands” several hundred encodings
  - Most important
    - ascii (corresponds to the 7-bit ASCII standard)
    - **utf-8** (usually your best bet for data from the Web)
    - latin-1
      - straight-forward interpretation of the 8-bit extended ASCII
      - never throws a “cannot decode” error
      - no guarantee that it read things the right way

Python and Encodings

- If Python tries to read a file and cannot decode, it throws a decoding exception and terminates execution
- We will learn about exceptions and how to handle them soon.
- For the time being: Write code that tells you where the problem is (e.g. by using line-numbers) and then fix the input.
- Usually, the presence of decoding errors means that you read the file in the wrong encoding

Using the os-module

- With the os-module, you can obtain greater access to the file system
    - Here is code to get the files in a directory

```
import os

def list_files(dir_name):
    files = os.listdir(dir_name)
    for my_file in files:
        print(my_file, 
os.path.getsize(dir_name+"/"+my_file))

list_files(“Example")
```

Using the os-module

Get a list of file names in the directory

```
import os

def list_files(dir_name)
    files = os.listdir(dir_name)
    for my_file in files:
        print(my_file,
os.path.getsize(dir_name+"/"+my_file))

list_files(“Example")
```

Use the os-module

```
import os

def list_files(dir_name):
    files = os.listdir(dir_name)
    for my_file in files:
        print(my_file,
os.path.getsize(dir_name+"/"+my_file))

list_files(“Example")
```

Creating the path name to the file

Use the os-module

```
import os

def list_files(dir_name):
    files = os.listdir(dir_name)
    for my_file in files:
        print(my_file,
os.path.getsize(dir_name+"/"+my_file))

list_files(“Example")
```

Gives the size of the file in bytes

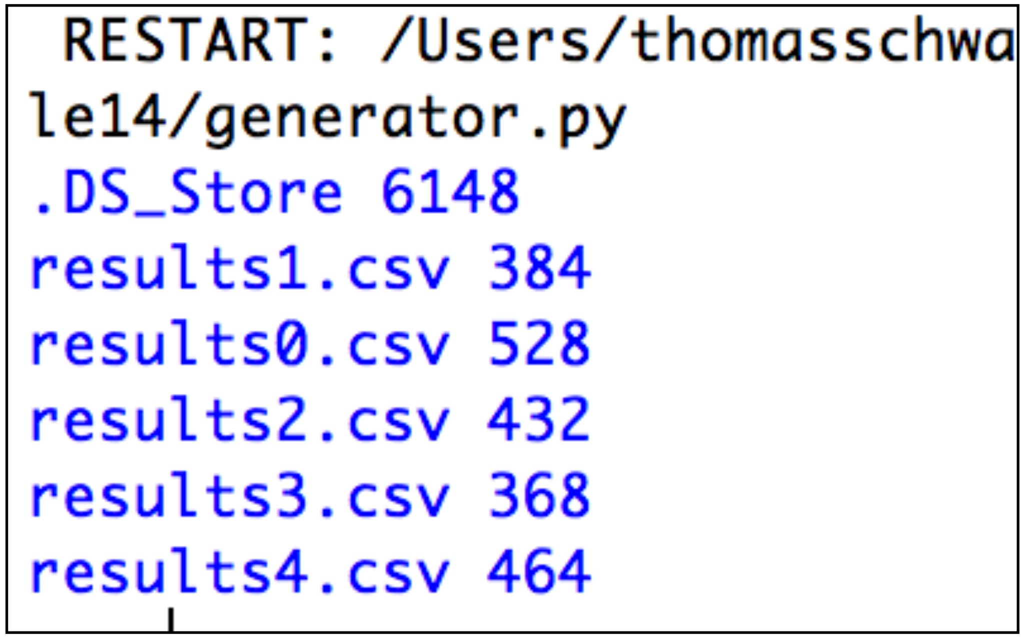
Use the os-module

```
import os

def list_files(dir_name):
    files = os.listdir(dir_name)
    for my_file in files:
        print(my_file,
os.path.getsize(dir_name+"/"+my_file))

list_files(“Example")
```

List and

Use the os-module

- Output:
    - Note the Mac-trash file



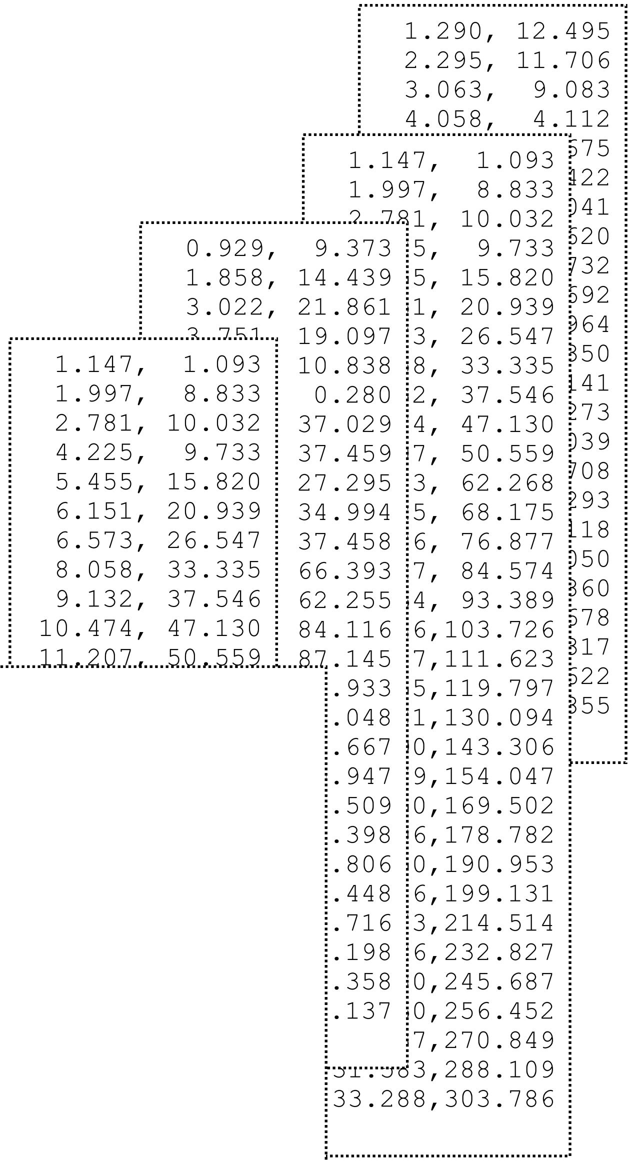
```
 RESTART: /Users/thomasschwa
le14/generator.py
.DS_Store 6148
results1.csv 384
results0.csv 528
results2.csv 432
results3.csv 368
results4.csv 464
```

Use the os-module

- Using the listing capability of the os-module, we can process all files in a directory
    - To avoid surprises, we best check the extension
    - Assume a function `process_a_file`
        - Our function opens a comma-separated (.csv) file
        - Calculates the average of the ratios of the second over the first entries

Use the os-module

- The process_a_file takes the file-name
    - Calculates the average ratio



```
 1.147,  1.093
 1.997,  8.833
 2.781, 10.032
 4.225,  9.733
 5.455, 15.820
 6.151, 20.939
 6.573, 26.547
 8.058, 33.335
 9.132, 37.546
10.474, 47.130
11.207, 50.559
```

```
0.929,  9.373
1.858, 14.439
3.022, 21.861
3.751, 19.097
       10.838
        0.280
       37.029
       37.459
       27.295
       34.994
       37.458
       66.393
       62.255
       84.116
       87.145
         .933
         .048
         .667
         .947
         .509
         .398
         .806
         .448
         .716
         .198
         .358
         .137
```

```
 1.147,  1.093
 1.997,  8.833
 2.781, 10.032
      5,  9.733
      5, 15.820
      1, 20.939
      3, 26.547
      8, 33.335
      2, 37.546
      4, 47.130
      7, 50.559
      3, 62.268
      5, 68.175
      6, 76.877
      7, 84.574
      4, 93.389
      6,103.726
      7,111.623
      5,119.797
      1,130.094
      0,143.306
      9,154.047
      0,169.502
      6,178.782
      0,190.953
      6,199.131
      3,214.514
      6,232.827
      0,245.687
      0,256.452
      7,270.849
31.583,288.109
33.288,303.786
```

```
1.290, 12.495
2.295, 11.706
3.063,  9.083
4.058,  4.112
          575
          422
          041
          620
          732
          592
          964
          850
          141
          273
          039
          708
          293
          118
          050
          860
          578
          817
          622
          855
```

```
def process_a_file(file_name):
    with open(file_name, "r") as infile:
        suma = 0
        nr_lines = 0
        for line in infile:
            nr_lines+=1
            array = line.split(',')
            suma+= float(array[1])/float(array[0])
    return suma/nr_lines
```

Use the os-module

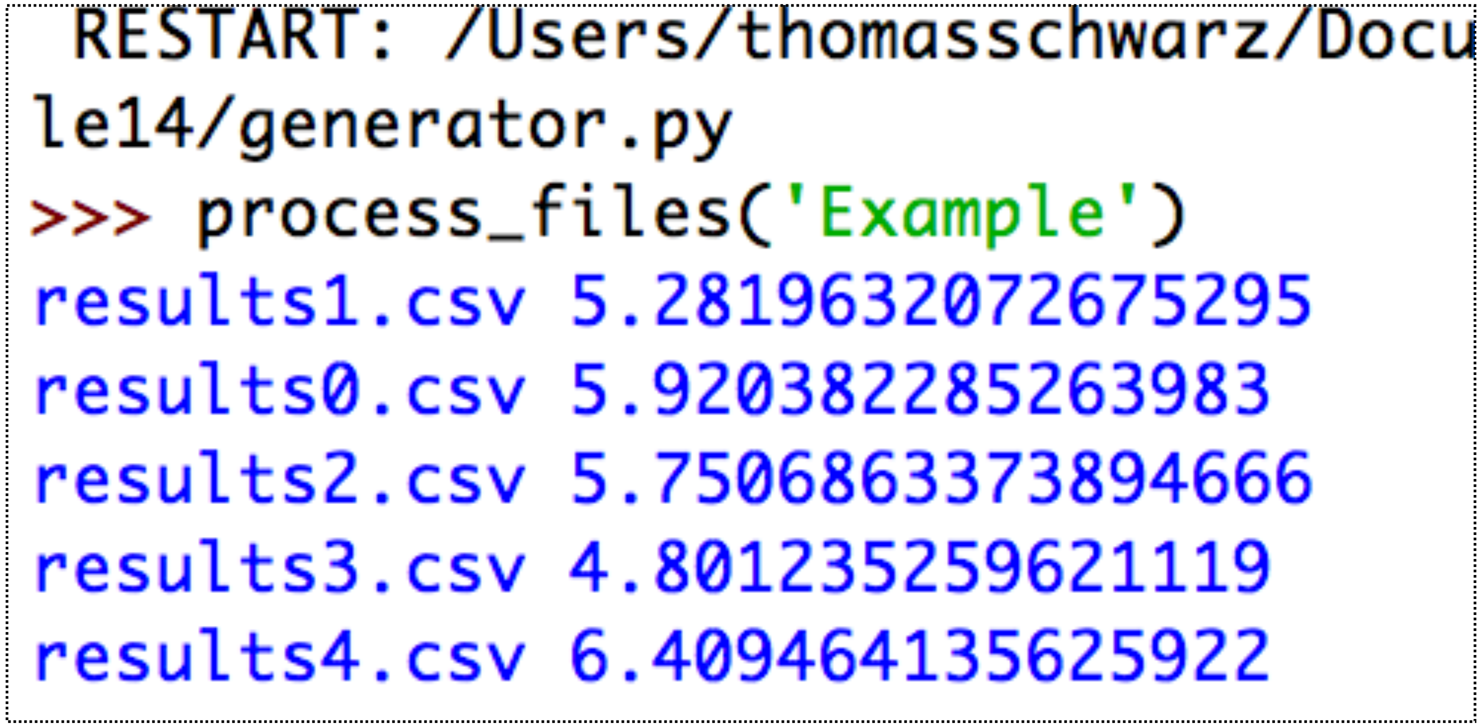
- To process the directory
  - Get the file names using os
  - For each file name:
    - Check whether the file name ends with .csv
    - Call the process_a_file function
    - Print out the result

Use of the os-module

```
def process_files(dir_name):
    files = os.listdir(dir_name)
    for my_file in files:
        if my_file.endswith('.csv'):
            print(my_file, process_a_file(
                           “Example/{}”.format(my_file)))
```

Using format to create the file name

Use of the os-module



```
 RESTART: /Users/thomasschwarz/Docu
le14/generator.py
>>> process_files('Example')
results1.csv 5.2819632072675295
results0.csv 5.920382285263983
results2.csv 5.7506863373894666
results3.csv 4.801235259621119
results4.csv 6.409464135625922
```

Encodings

- Whenever you see strings:
    - Think about encoding and decoding
        - Example: the ë
            - `'ë'.encode('utf-8').decode('latin-1')`
        - gives
            - `'Ã«'`
- Mixing encodings often creates chaos

Encodings

- Python is very good at guessing encodings
    - Do not guess encodings
        - E.g.: Processing html: read the http header:
            - `Content-Type: text/html; charset=utf-8`
    - If you need to guess, there is a module for it:
        - `chardet.detect(some_bytes)`

Encodings

- Thinking about encoding and decoding string allows easy internationalization

Bytearrays

- On (rare) occasions, you might want to work with bytes directly
    - Read the file in binary mode
    - Bytearray allows you to manipulate directly binary data
        - bytes have range 0-255
    - `content = bytearray(infile.read())`

Exceptions

Exceptions

- There are two approaches to living life as a religious:
    - Before you do anything, you ask for permission
        - Strengthens humility and denial of self
    - Do something and then ask for pardon
        - Strengthens your Ego too much, but makes it easier on the superior

- Similarly: There are two approaches to the risks of live:
    - Make sure you are prepared for anything
    - Just live your life and deal with the consequences of your errors.

- In programming, Python tends to fall squarely into the second category
    - But it makes more sense than in real life

Exceptions

- *RAISING AN EXCEPTION* interrupts the flow of the program
- *HANDLING AN EXCEPTION* puts the program flow back on track or deals with an error situation
  - Such as out of memory, file cannot be found, CPU illegal instruction error, division by zero, overflow, …

Python Philosophy

Socrates scores, got a beautiful cross from Archimedes. The Germans are disputing it.
Hegel is arguing that the reality is merely an a priori adjunct of non-naturalistic ethics.
Kant, via the categorical imperative, is holding that, ontologically, it exists only in the imagination.
And Marx is claiming it was offside.

Philosopher’s Football

- Handle the common case.
    - And deal with the exceptions.

C, Java, C++ Philosophy

- C: check before you assume
- Java, C++: Use exceptions to handle bad situations
- Python: Use exceptions for the not so ordinary

Python

- If an instruction or block of instruction can cause an error, put it in a *try block*.

```
try:
    int(string)
```

Converts the string into an integer

Notice that we are not using the result of the conversion, we just attempt the conversion

Python Exceptions

- Then afterwards, *handle the exception.*
  - You *should,* but are not required to specify the possible offending exception

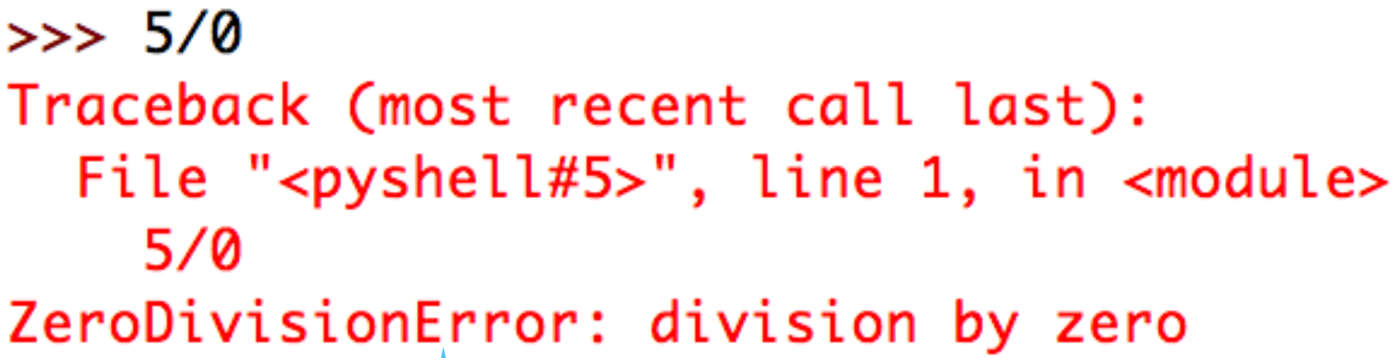
```
try:
     int(string)
except ValueError:
     print(“Conversion error”)
```

If the conversion fails, a ValueError is thrown

This block handles the exception

Python Exceptions

- How do you find which error is thrown:
  - You can cause the error and see what type of error it is
  - You can look it up



```
>>> 5/0
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    5/0
ZeroDivisionError: division by zero
```

Division by zero creates a ZeroDivisionError

Python Exceptions

- Putting things together: Testing whether a string represents an integer

```
def is_int(string):
    try:
        int(string)
        return True
    except:
        return False
```

Try out the conversion

Python Exceptions

- Putting things together: Testing whether a string represents an integer

```
def is_int(string):
    try:
        int(string)
        return True
    except:
        return False
```

Try out the conversion

It worked:
We return True

Python Exceptions

- Putting things together: Testing whether a string represents an integer

```
def is_int(string):
    try:
        int(string)
        return True
    except:
        return False
```

Try out the conversion

It did NOT work:
An exception is thrown
We return FALSE

Python Exceptions

- As you can see from this example, the moment an exception is thrown, we jump to the exception handler.

Python Exceptions

- When to use exceptions and when to use if
  - Recall: Using `if` is defensive programming
  - Recall: Using exceptions amounts to the same degree of safety, but is offensive
- Rule of thumb:
  - If exceptions are raised infrequently, then use them

Python Exceptions

- Let’s make some timing experiments
  - Define two functions that square all elements in a list, if the elements are integers.

```
def square_list(lista):
    result = []
    for element in lista:
        if element.isdigit():
            result.append(int(element)**2)
```

```
def square_list2(lista):
    result = []
    for element in lista:
        try:
           result.append(int(element)**2)
        except:
            pass
```

Python Exceptions

- The pass instruction:
  - When Python expects a statement, but we don't have one:
    - Just use `pass`
      - The No-Operation instruction

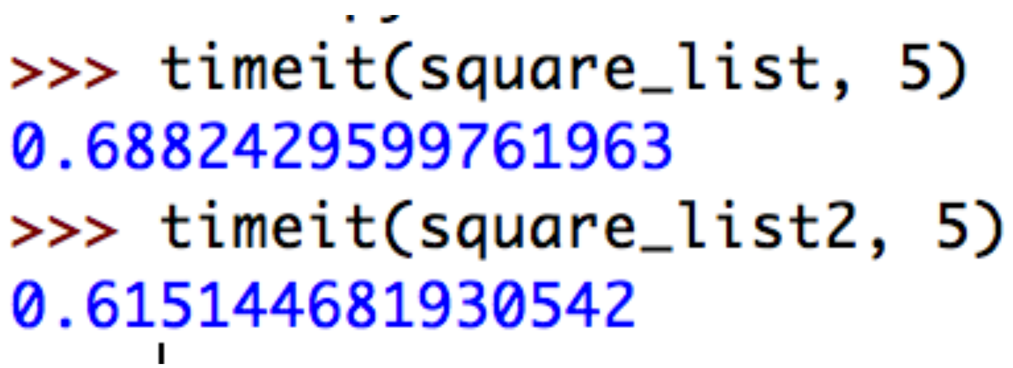
Python Exceptions

- Recall how to use the time-module to obtain the CPU (wall-clock) time
- We use this to measure execution time
    - First a list that only contains integers

```
def timeit(function, trials):
    lista = [str(i) for i in range(1000000)]
    count = 0
    for _ in range(trials):
        start = time.time()
        lista2 = function(lista)
        count += time.time()-start
    return count/trials
```

Python Exceptions

- Result: Exceptions are somewhat faster

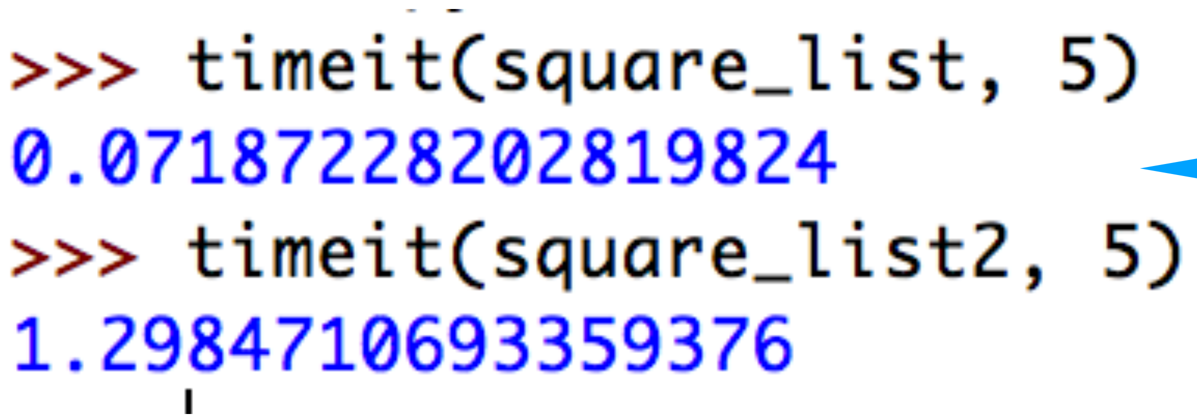


```
>>> timeit(square_list, 5)
0.6882429599761963
>>> timeit(square_list2, 5)
0.61514468193054
```

Python Exceptions

- What if none of the list elements are integers:

```
def timeit(function, trials):
    lista = ["a"+str(i) for i in range(1000000)]
    count = 0
    for _ in range(trials):
        start = time.time()
        lista2 = function(lista)
        count += time.time()-start
    return count/trials
```



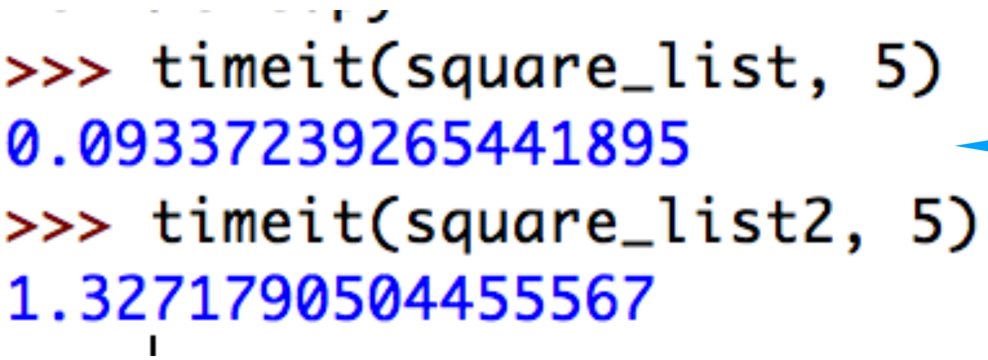
```
>>> timeit(square_list, 5)
0.07187228202819824
>>> timeit(square_list2, 5)
1.298471069335937б
```

Exceptions are much slower

Python Exceptions

- What about if the letter is at the end

```
def timeit(function, trials):
    lista = [str(i)+"a" for i in range(1000000)]
    count = 0
    for _ in range(trials):
        start = time.time()
        lista2 = function(lista)
        count += time.time()-start
    return count/trials
```



```
>>> timeit(square_list, 5)
0.09337239265441895
>>> timeit(square_list2, 5)
1.327179050445567
```

Exceptions are still much slower

Self Test

- Define a function that calculates the geometric mean of two numbers.
- Use an exception to deal with a ValueError, arisen by taking the square-root of a negative number
    - Here is the if-version. We return `None` if there is no mean.

```
def geo(x, y):
    if x*y > 0:
        return math.sqrt(x*y)
    return None
```

Self Test Solution

```
def geoe(x,y):
    try:
        return math.sqrt(x*y)
    except ValueError:
        return None
```

Multiple Exceptions

- We can write an exception handler that handles all the exceptions
  - This is discouraged since there are just too many exceptions that can occur
    - such as out-of-memory, system-error, keyboard-interrupt …
  - In this case, the except clause specifies no exception

```
try:
   accum += 1/n
except:
    print(“something bad happened“
```

No exception specified
Handler handles
everything

Multiple Exceptions

- Normally, you want to specify which exceptions you are handling
- You can specify several exception handles by repeating the exception clause
- Or you can handle a list of exceptions

```
def test():
    try:
        f = open("none.txt")
        block = f.read(256)
    except IOError:
        print("something happened when reading the file")
    except EOFError:
        print("ran out of file")
    except (KeyboardInterrupt, ValueError):
        print("something strange happened")
```

The parentheses are necessary

Cleaning Up

- Sometimes you need to make sure that failure-prone code cleans up
- Use the `finally` clause
    - Guaranteed to be executed
        - Even with return statements
        - Even when exceptions are raised

Example for `finally` clause

- If we open a file without the if-clause, we are morally obliged to close it
  - Let’s say, if you have a long-running process that only needs a file for a little time, you should not hog the file and prevent others from accessing it.

Example for `finally` clause

```
def harmonic(filename):
    """
    Assumes that the elements in the file are numbers.
    We return the harmonic mean of the numbers.
    """
    count = 0
    accumulator = 0
    try:
        infile = open(filename, encoding="utf-8")
        for line in infile:
            for words in line.split():
                accumulator += 1/int(words)
                count += 1
        return count/accumulator
    except ZeroDivisionError:
        print("saw a zero")
        return 1000000000
    except ValueError:
        print("saw a non-integer")
        return 0
    finally:
        print("I am done and closing the file")
        infile.close()
```

Return in the try block

Return in the handler

But finally is guaranteed to run before any of the returns

Raising exceptions

- You can also raise your own exception
  - You can even define your own exceptions when you have understood classes
  - Just say: `raise ValueError`
  - or whatever the exception is that you want to raise.

Self Test

- Recall that the finally clause is always executed.
- What is the output of the following code

```
def raising():
    try:
        raise ValueError
    except ValueError:
        return 0
    finally:
        return 1
```

Answer

- The functions returns 1
    - The exception is raised and control passes to the exception handler
    - Before the exception handler can return, the finally clause is executed
    - And that one returns 1

Multiple Exceptions

- It is common that Python code throws multiple exceptions
  - Can list different exceptions using a tuple and handle them all

```
try:
   client_obj.get_url(url)
except (URLError, ValueError, SocketTimeout):
   client_obj.remove_url(url)
```

  - Or write different exception handlers

```
try:
   client_obj.get_url(url)
except (URLError, ValueError):
   client_obj.remove_url(url)
except SocketTimeout:
   client_obj.handle_url_timeout(url)
```

Handles to Exceptions

- Exceptions are classes that have methods
- To gain access use the `as` keyword

```
try:
   f = open(filename)
except OSError as e:
   if e.errno == errno.ENOENT:
      print('file not found')
   elif e.errno == errno.EACCES:
      print('permission denied')
   else:
      print('unexpected error')
```

Multiple Exceptions

- More than one exception can be triggered
  - The first matching exception handler will handle, even if a more specific exception handler is available

```
try:
   f = open(a_missing_file)
except OSError:
   print('it failed')
except FileNotFoundError:
   print('File not found')
```

  - prints out 'it failed'

Multiple Exceptions

- Exceptions are in a hierarchy

```
try:
    …
except Exception as e:
    …
    print(e)
```

  - catches all exceptions except SystemExit, KeyboardInterrupt, GeneratorExit
  - If you want to catch those, change Exception to BaseException

Creating Custom Exceptions

- To create a new exception, just define a class that derives from Exception

```
class NetworkError(Exception):
   pass
class TimeoutError(NetworkError):
   pass
```

Creating Custom Exceptions

- If your custom exception overrides the constructor
  - Make sure you call the exception class constructor

```
class CustomError(Exception):
   def __init__(self, message, status):
      self.message = message
      self.status = status
```

  - Parts of Python and libraries except all exceptions to have an .args attribute, that will be provided by calling the super

Chaining Exceptions

- Raise an exception in response to catching a different exception, but include information about both exceptions in the traceback

```
def example():
   try:
      int('N/A')
   except ValueError as e:
      raise RuntimeError('A parsing error occured') from e
```

Assertions

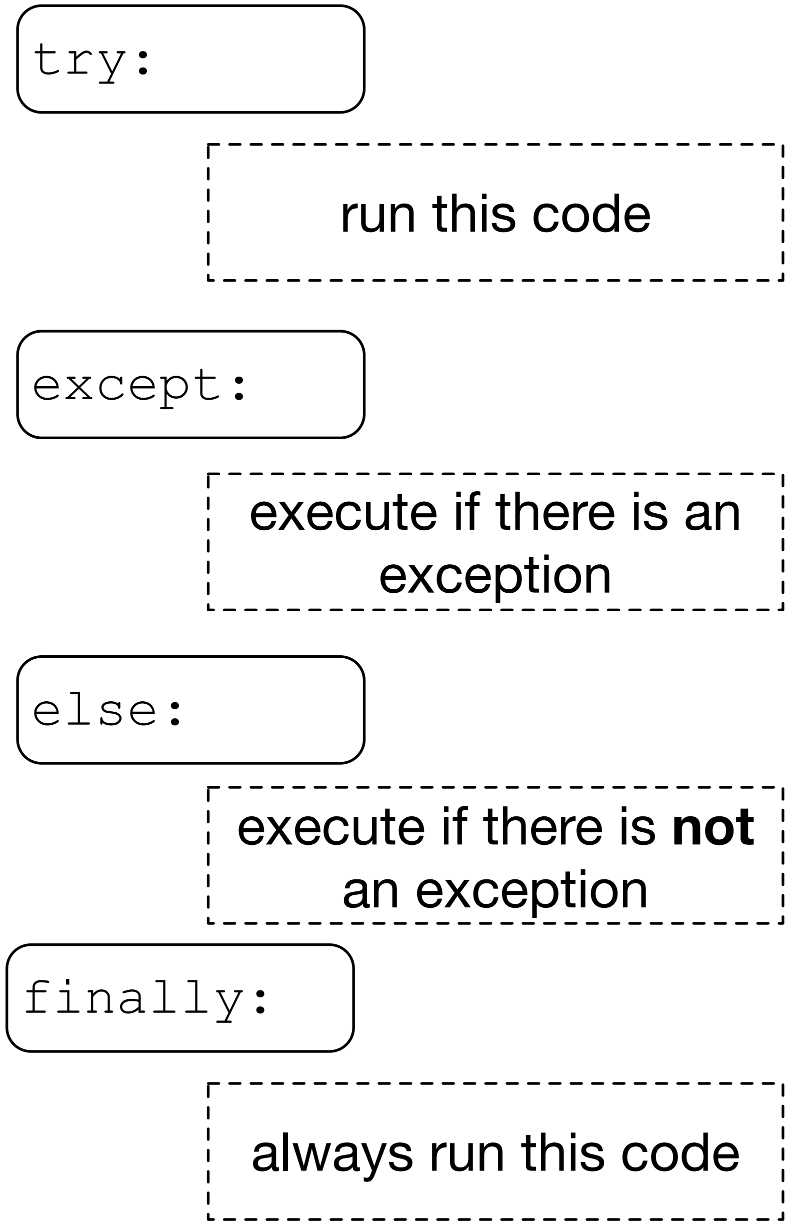
- To prevent error conditions, can use assertions
  - E.g.: your code only runs on a linux machine

```
import sys

assert ('linux' in sys.platform),
        'this code runs on linus only')
```

  - If the condition is violated, throws an AssertionError
  - But the assert statements are optimized away when

Else Statement

- Else block after a try block is executed only if no exception was raised



`try:`

run this code

`except:`

execute if there is an exception

`else:`

execute if there is **not** an exception

`finally:`

always run this code

-

Else Statement

- Exceptions in the else block would not be caught by the current try block

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except OSError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

Exercises

- The following code is potentially buggy.

```
info = [{'score': 3, 'confidence': 2},
        {'score': -1, 'confidence': 4},
        {'score': 1, 'confidence': 4},
        {'confidence': 0}]

def get_total_score(info):
   total = 0
   for item in info:
      total += item['score']
   return total

get_total_score(info)
```

Solutions

```
def get_total_score(info):
    total = 0
    number_of_items = 0
    for item in info:
        try:
            total += item['score']
        except KeyError:
            pass
        else:
            number_of_items += 1
    return total/number_of_items

print(get_total_score(info))
```

Exercises

- The following code is potentially buggy.

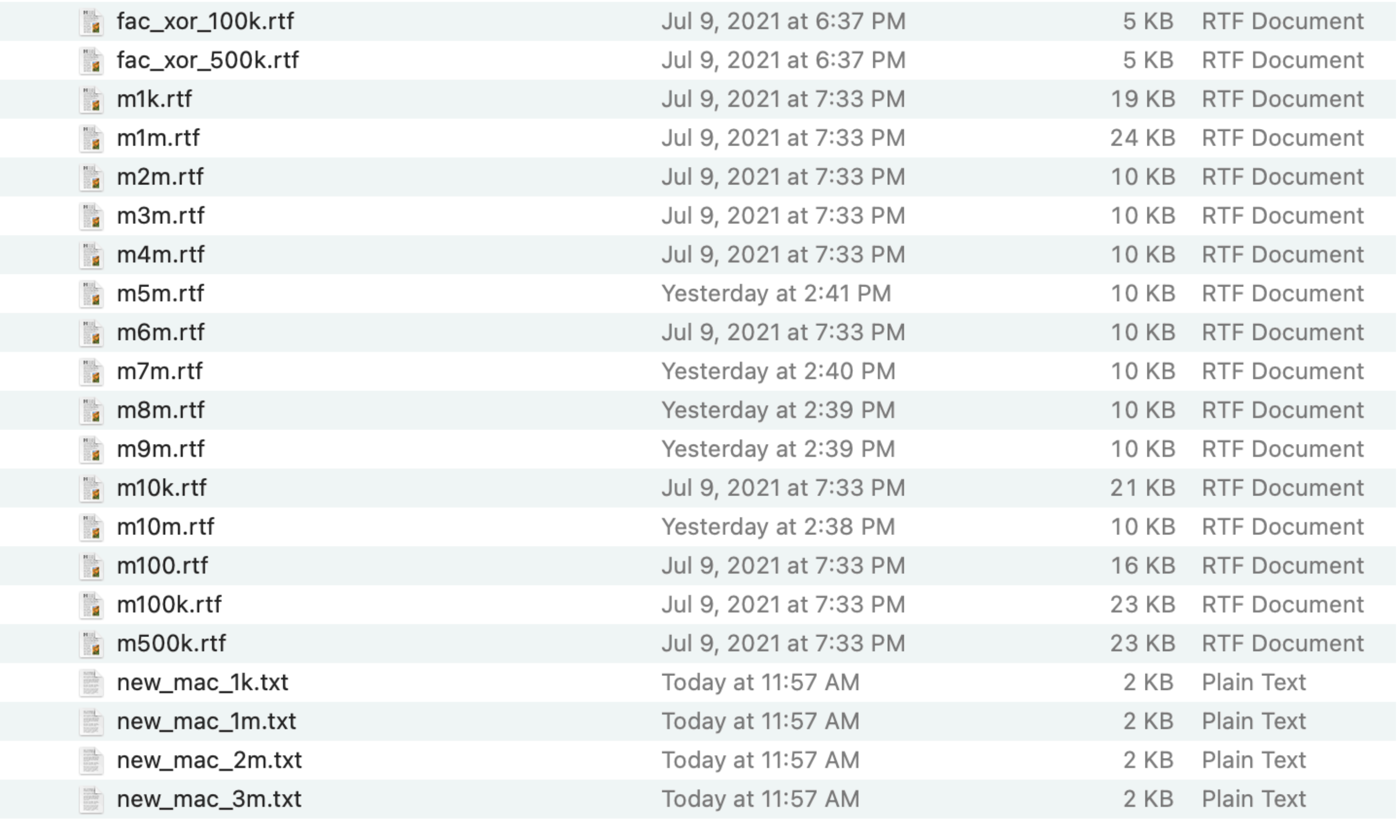
```
import os

def check(directory):
    for file_name in os.listdir(directory):
        with open(file_name) as infile:
           nr = len(infile.readlines())
           print(file_name, nr)
```

Solutions

```
import os

def check(directory):
    for file_name in os.listdir(directory):
        try:
            with open(file_name) as infile:
                nr = len(infile.readlines())
                print(file_name, nr)
        except UnicodeDecodeError:
            print('unicode decode error in', file_name)
        except IsADirectoryError:
            print(f'{file_name} is a directory')
```

Use Case

Use Case

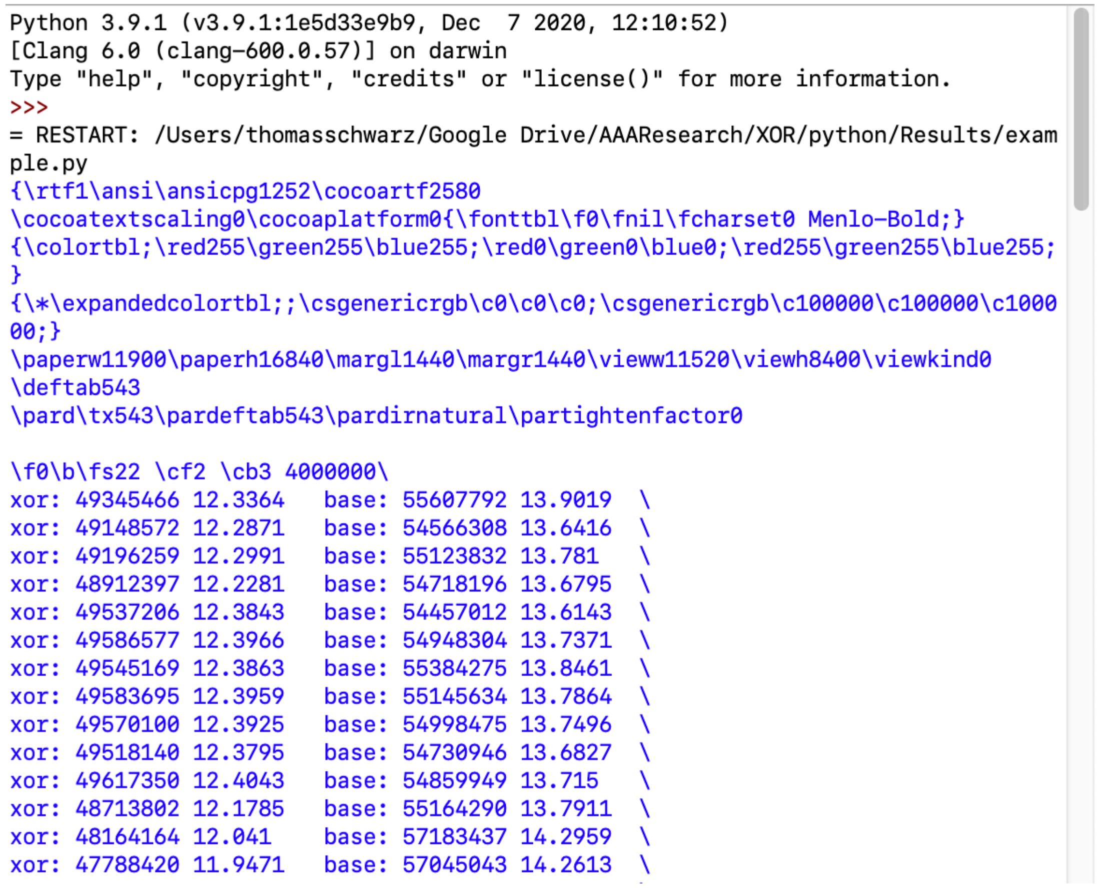
- Given experimental data in several files, generate statistics: mean, median, standard deviation, min, max
- First, need to read and understand the files
  - 

fac_xor_100k.rtf	Jul 9, 2021 at 6:37 PM	5 KB	RTF Document
fac_xor_500k.rtf	Jul 9, 2021 at 6:37 PM	5 KB	RTF Document
m1k.rtf	Jul 9, 2021 at 7:33 PM	19 KB	RTF Document
m1m.rtf	Jul 9, 2021 at 7:33 PM	24 KB	RTF Document
m2m.rtf	Jul 9, 2021 at 7:33 PM	10 KB	RTF Document
m3m.rtf	Jul 9, 2021 at 7:33 PM	10 KB	RTF Document
m4m.rtf	Jul 9, 2021 at 7:33 PM	10 KB	RTF Document
m5m.rtf	Yesterday at 2:41 PM	10 KB	RTF Document
m6m.rtf	Jul 9, 2021 at 7:33 PM	10 KB	RTF Document
m7m.rtf	Yesterday at 2:40 PM	10 KB	RTF Document
m8m.rtf	Yesterday at 2:39 PM	10 KB	RTF Document
m9m.rtf	Yesterday at 2:39 PM	10 KB	RTF Document
m10k.rtf	Jul 9, 2021 at 7:33 PM	21 KB	RTF Document
m10m.rtf	Yesterday at 2:38 PM	10 KB	RTF Document
m100.rtf	Jul 9, 2021 at 7:33 PM	16 KB	RTF Document
m100k.rtf	Jul 9, 2021 at 7:33 PM	23 KB	RTF Document
m500k.rtf	Jul 9, 2021 at 7:33 PM	23 KB	RTF Document
new_mac_1k.txt	Today at 11:57 AM	2 KB	Plain Text
new_mac_1m.txt	Today at 11:57 AM	2 KB	Plain Text
new_mac_2m.txt	Today at 11:57 AM	2 KB	Plain Text
new_mac_3m.txt	Today at 11:57 AM	2 KB	Plain Text

Understanding the File

- We want to extract data from the rtf files
    - Which is a special format with some metadata
    - So, we open up a file and read its contents:

```
with open('m4m.rtf') as infile:
    for line in infile:
        print(line.strip())
```



```
Python 3.9.1 (v3.9.1:1e5d33e9b9, Dec  7 2020, 12:10:52)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: /Users/thomasschwarz/Google Drive/AAAResearch/XOR/python/Results/exam
ple.py
{\rtf1\ansi\ansicpg1252\cocoartf2580
\cocoatextscaling0\cocoaplatform0{\fonttbl\f0\fnil\fcharset0 Menlo-Bold;}
{\colortbl;\red255\green255\blue255;\red0\green0\blue0;\red255\green255\blue255;
}
{\*\expandedcolortbl;;\csgenericrgb\c0\c0\c0;\csgenericrgb\c100000\c100000\c1000
00;}
\paperw11900\paperh16840\margl1440\margr1440\vieww11520\viewh8400\viewkind0
\deftab543
\pard\tx543\pardeftab543\pardirnatural\partightenfactor0

\f0\b\fs22 \cf2 \cb3 4000000\
xor: 49345466 12.3364   base: 55607792 13.9019  \
xor: 49148572 12.2871   base: 54566308 13.6416  \
xor: 49196259 12.2991   base: 55123832 13.781   \
xor: 48912397 12.2281   base: 54718196 13.6795  \
xor: 49537206 12.3843   base: 54457012 13.6143  \
xor: 49586577 12.3966   base: 54948304 13.7371  \
xor: 49545169 12.3863   base: 55384275 13.8461  \
xor: 49583695 12.3959   base: 55145634 13.7864  \
xor: 49570100 12.3925   base: 54998475 13.7496  \
xor: 49518140 12.3795   base: 54730946 13.6827  \
xor: 49617350 12.4043   base: 54859949 13.715   \
xor: 48713802 12.1785   base: 55164290 13.7911  \
xor: 48164164 12.041    base: 57183437 14.2959  \
xor: 47788420 11.9471   base: 57045043 14.2613  \
```

Understanding the File

- First thing: 'rtf' is good because we do not need to struggle with encoding
- Second: We want to extract the data from the second and fifth column and get statistics about them
- Third: The data is organized into files and the file name gives the parameter. The parameter also appear in the nineth line.

```
with open('m4m.rtf') as infile:
    for _ in range(9):
        line = infile.readline()
    if '4000000' in line:
        print(line)
```

Checking the File

- To open up all the files, we use a for loop
  - This gives us more control then using the os-interface because files might be added to the directory
  - Trick: Just put the part of the filename into a list that changes

Checking the File

- We also want to ensure that the file name and the putative parameter are the same.
  - Write the parameters and the filenames into a list
  - Then in a for, loop over the zip of the two lists

Checking the File

```
numbers = [100, 1000, 10000, 100000, 500000, 10**6, 2*10**6,
3*10**6, 4*10**6,
           5*10**6, 6*10**6, 7*10**6, 8*10**6, 9*10**6,
10*10**6]
for filename, number in zip(['100','1k', '10k',
   '100k', '500k', '1m', '2m', '3m', '4m', '5m',
   '6m', '7m', '8m', '9m', '10m'],numbers):
    filename = 'm'+filename+'.rtf'
    with open(filename) as infile:
        for _ in range(9):
            line = infile.readline()
        if str(number) in line:
            print(f'Processing {filename}.')
        else:
            print(f'Error in {filename}')
```

Extracting the Data

- After the next line, there is data

  -
    ```
    xor: 49345466 12.3364 base: 55607792 13.9019   \
    xor: 49148572 12.2871 base: 54566308 13.6416   \
    xor: 49196259 12.2991 base: 55123832 13.781 \
    xor: 48912397 12.2281 base: 54718196 13.6795   \
    xor: 49537206 12.3843 base: 54457012 13.6143   \
    xor: 49586577 12.3966 base: 54948304 13.7371   \
    ```
- Extract the second and the fifth column
- This uses split

```
for line in infile:
    contents = line.strip().split()
```

Extracting the Data

- The result is an array with substrings:

```
['xor:', '721', '7.21', 'base:', '1188', '11.88', '\\']
['xor:', '761', '7.61', 'base:', '1192', '11.92', '\\']
['xor:', '754', '7.54', 'base:', '1192', '11.92', '\\']
['xor:', '705', '7.05', 'base:', '1008', '10.08', '\\']
['xor:', '640', '6.4', 'base:', '1047', '10.47', '\\']
['xor:', '608', '6.08', 'base:', '1049', '10.49', '\\']
['xor:', '658', '6.58', 'base:', '1049', '10.49', '\\']
['xor:', '679', '6.79', 'base:', '1049', '10.49', '\\']
```

- You might notice the escaped back-slash at the end

Extracting the Data

- We convert the substrings to ints and store them in an array each

```
xor, base = [], []
for line in infile:
    contents = line.strip().split()
    try:
        xor.append(int(contents[1]))
        base.append(int(contents[4]))
    except:
        print(line, 'is causing a problem')
```

Processing the Data

- Now we process these numbers
    - We are given an array
    - We want to obtain min, max, mean, median, standard deviation
    - Some of this are built in functions

Processing the Data

- Can also use sum on an array

```
def process(numbers):
    mymin = min(numbers)
    mymax = max(numbers)
    mean = sum(numbers)/len(numbers)
```

- Standard Deviation is the average square of the difference between value and mean

```
stddev = sum([(x-mean)**2 for x in numbers])/len(numbers)
```

Processing the Data

- Median is the middle value if the number of elements is odd
    - and the mean of the two middle numbers if the number of elements is even

```
if len(numbers)%2:  #odd number of elements
        median = numbers[len(numbers)//2]
    else:   #even number of elements
        median = 0.5*(numbers[len(numbers)//
2-1]+numbers[len(numbers)//2])
```

- Recall: // is integer (or floor) division

Processing the Data

- We use a tuple to return all these values

```
return mymin, mymax, mean, stddev, median
```

Output the Results

- Now we need to write the results into a file
  - Let's open and close it manually

```
outfile = open('results.csv', 'w')

…

outfile.close()
```

Output the Results

- We write the results into a csv file
- We can just use print, though sometimes formatting is more appropriate
  - Outside the loop

```
print('number', 'xmymin', 'xmymax', 'xmean', 'xstdev', 'xmedian',
               'bmymin', 'bmymax', 'bmean', 'bstdev', 'bmedian',
       sep=',', file=outfile)
```

  - Inside the loop

```
print(number, xmymin/number, xmymax/number, xmean/number, xstdev/number, xmedian/number,
              bmymin/number, bmymax/number, bmean/number, bstdev/number, bmedian/number,
      sep=',', file=outfile)
```

Output the Results

- The result can be opened up with a default csv reader

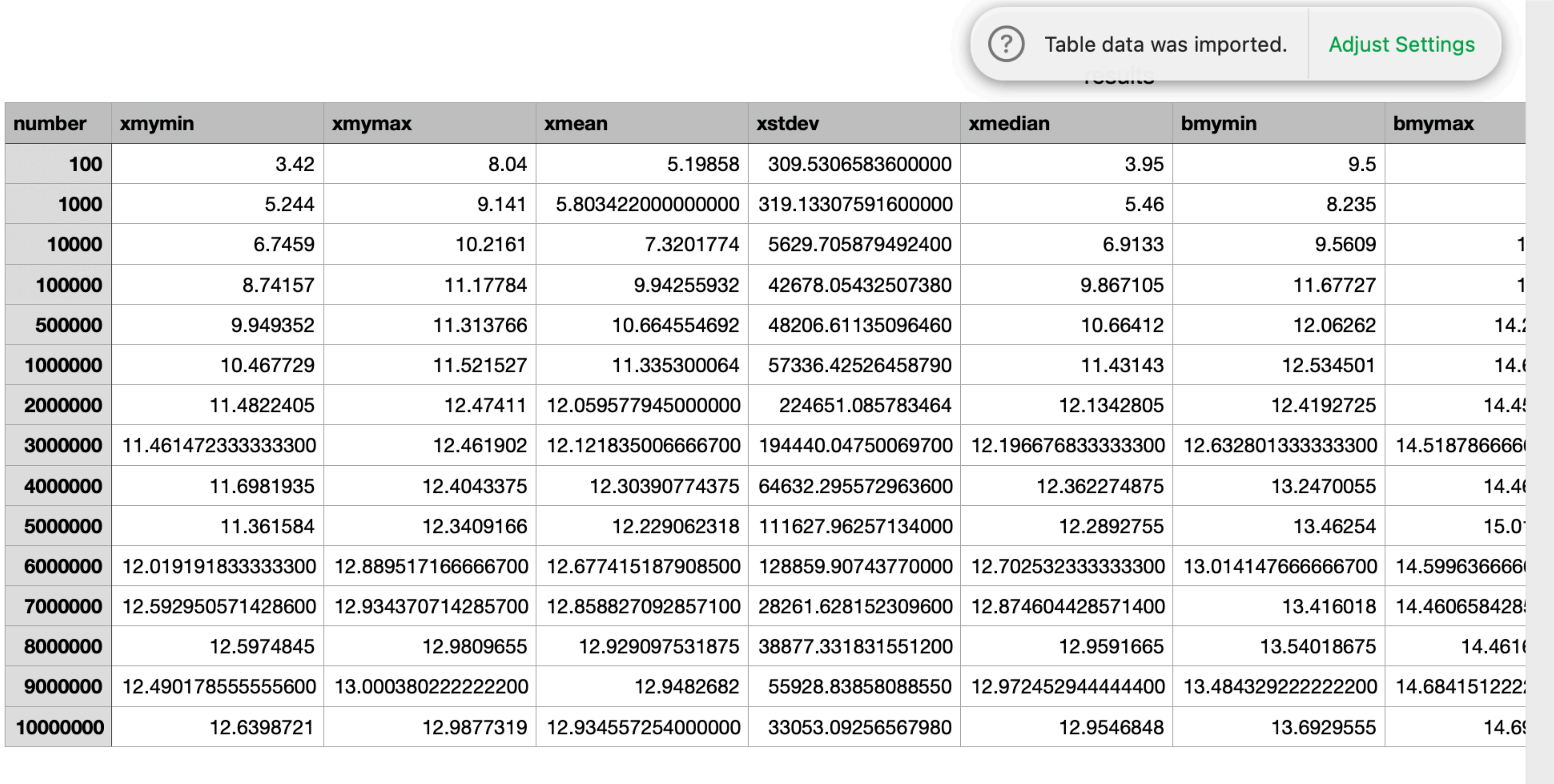
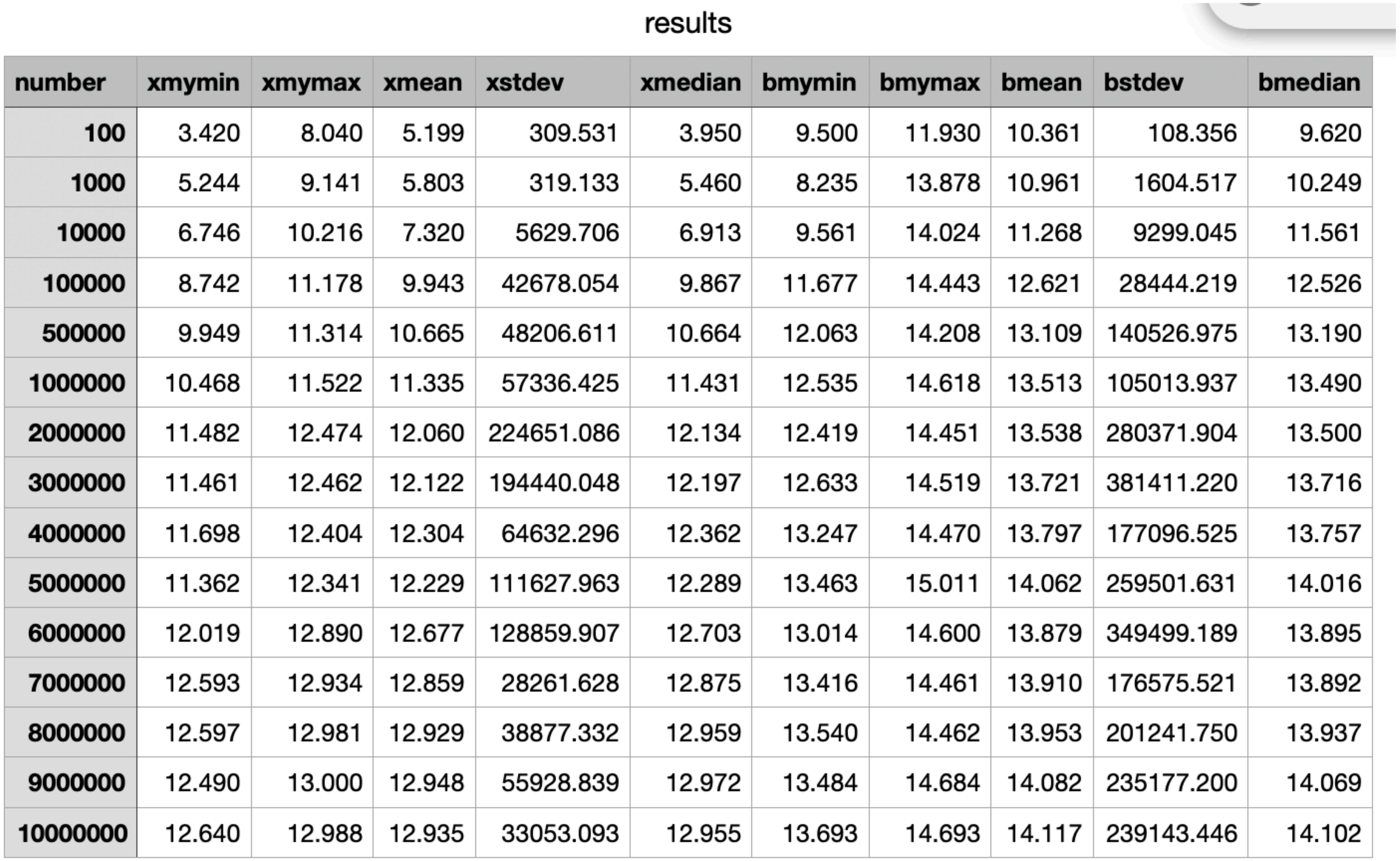


Table data was imported.

number	xmymin	xmymax	xmean	xstdev	xmedian	bmymin	bmymax
100	3.42	8.04	5.19858	309.5306583600000	3.95	9.5	
1000	5.244	9.141	5.803422000000000	319.13307591600000	5.46	8.235	
10000	6.7459	10.2161	7.3201774	5629.705879492400	6.9133	9.5609	1
100000	8.74157	11.17784	9.94255932	42678.05432507380	9.867105	11.67727	1
500000	9.949352	11.313766	10.664554692	48206.61135096460	10.66412	12.06262	14.2
1000000	10.467729	11.521527	11.335300064	57336.42526458790	11.43143	12.534501	14.6
2000000	11.4822405	12.47411	12.059577945000000	224651.085783464	12.1342805	12.4192725	14.45
3000000	11.461472333333300	12.461902	12.121835006666700	194440.04750069700	12.196676833333300	12.632801333333300	14.518786666
4000000	11.6981935	12.4043375	12.30390774375	64632.295572963600	12.362274875	13.2470055	14.46
5000000	11.361584	12.3409166	12.229062318	111627.96257134000	12.2892755	13.46254	15.01
6000000	12.019191833333300	12.889517166666700	12.677415187908500	128859.90743770000	12.702532333333300	13.014147666666700	14.599636666
7000000	12.592950571428600	12.934370714285700	12.858827092857100	28261.628152309600	12.874604428571400	13.416018	14.460658428
8000000	12.5974845	12.9809655	12.929097531875	38877.331831551200	12.9591665	13.54018675	14.4616
9000000	12.490178555555600	13.000380222222200	12.9482682	55928.83858088550	12.972452944444400	13.484329222222200	14.684151222
10000000	12.6398721	12.9877319	12.934557254000000	33053.09256567980	12.9546848	13.6929555	14.69

Output the Results

- Clearly, a format string is appropriate.



results

number	xmymin	xmymax	xmean	xstdev	xmedian	bmymin	bmymax	bmean	bstdev	bmedian
100	3.420	8.040	5.199	309.531	3.950	9.500	11.930	10.361	108.356	9.620
1000	5.244	9.141	5.803	319.133	5.460	8.235	13.878	10.961	1604.517	10.249
10000	6.746	10.216	7.320	5629.706	6.913	9.561	14.024	11.268	9299.045	11.561
100000	8.742	11.178	9.943	42678.054	9.867	11.677	14.443	12.621	28444.219	12.526
500000	9.949	11.314	10.665	48206.611	10.664	12.063	14.208	13.109	140526.975	13.190
1000000	10.468	11.522	11.335	57336.425	11.431	12.535	14.618	13.513	105013.937	13.490
2000000	11.482	12.474	12.060	224651.086	12.134	12.419	14.451	13.538	280371.904	13.500
3000000	11.461	12.462	12.122	194440.048	12.197	12.633	14.519	13.721	381411.220	13.716
4000000	11.698	12.404	12.304	64632.296	12.362	13.247	14.470	13.797	177096.525	13.757
5000000	11.362	12.341	12.229	111627.963	12.289	13.463	15.011	14.062	259501.631	14.016
6000000	12.019	12.890	12.677	128859.907	12.703	13.014	14.600	13.879	349499.189	13.895
7000000	12.593	12.934	12.859	28261.628	12.875	13.416	14.461	13.910	176575.521	13.892
8000000	12.597	12.981	12.929	38877.332	12.959	13.540	14.462	13.953	201241.750	13.937
9000000	12.490	13.000	12.948	55928.839	12.972	13.484	14.684	14.082	235177.200	14.069
10000000	12.640	12.988	12.935	33053.093	12.955	13.693	14.693	14.117	239143.446	14.102

Checking the Results

- Which of these columns does not make sense?
- Where is the error?