INPUTS – OUTPUTS FILES

Python Basics

OUTLINE

Standard input/output

Open function

Characters

Reading Data

Using “With”

File Types

- CSV
- JSON
- HTML/XML

Operating System (OS)

Accessing online files and crawling

BUILTINS

```
input(prompt)
```

Gets user input until the ENTER key is pressed; returns it as a string (without any newline). If there's a prompt string, this is printed to the current prompt line.

Example

```
name = input("What's your name? ")
print("Nice to meet you " + name + "!")
```

STANDARD INPUT/OUTPUT

Input reads stdin (usually keyboard), in the same way print writes to stdout (usually the screen).

Generally, when we move information between programs, or between programs and hardware, we talk about streams: tubes down which we can send data.

Stdin and stdout can be regarded as streams from the keyboard to the program, and from the program to the screen.

There's also a stderr where error messages from programs are sent: again, usually the screen by default.

STANDARD INPUT/OUTPUT

You can redirect these, for example, at the command prompt:

Stdin from file:

```
python a.py < stdin.txt
```

Stdout to overwritten file:

```
python a.py > stdout.txt
```

Stdout to appended file:

```
python a.py >> stdout.txt
```

Both:

```
python a.py < stdin.txt > stdout.txt
```

You can also pipe the stdout of one program to the stdin of another program using the pipe symbol "|" (SHIFT- backslash on most Windows keyboards)

OPEN

Reading and writing files is a real joy in Python, which makes a complicated job trivial.

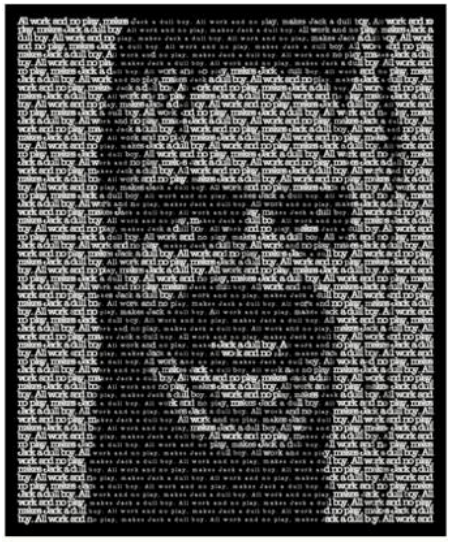
The builtin open function is the main method:

```
f = open("filename.txt")
for line in f:
        print(line)
f.close()
```

```
f = open("filename.txt")     # Whole file as string
```

Note the close function (not listed as a builtin). This is polite - it releases the file.

OPEN



To write:

```
a = []
for i in range(100):
    a.append("All work and no play makes Jack a dull boy ");
f = open("anotherfile.txt", 'w')
for line in a:
       f.write(line)
f.close()
```

LINE ENDINGS

With "write" you may need to write line endings.

The line endings in files vary depending on the operating system.

POSIX systems (Linux; MacOS; etc.) use the ASCII newline character, represented by the escape character `\n`.

Windows uses two characters: ASCII carriage return (`\r`) (which was used by typewriters to return the typing head to the start of the line), followed by newline.

You can find the OS default using the os library: `os.linesep`

But generally if you use `\n`, the Python default, Windows copes with it fine, and directly using `os.linesep` is advised against.

SEEK

It's usual to read an entire file.

However, if you want to jump within the file, use:

```
file.seek()
```

https://docs.python.org/3.3/tutorial/inputoutput.html#methods-of-file-objects

or

linecache: Random access to text lines

https://docs.python.org/3/library/linecache.html

BINARY VS TEXT FILES

The type of the file has a big effect on how we handle it.

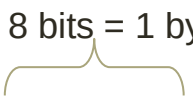
There are broadly two types of files: text and binary.

They are all basically ones and zeros; what is different is how a computer displays them to us, and how much space they take up.

BINARY VS. TEXT FILES

All files are really just binary 0 and 1 bits.

In 'binary' files, data is stored in binary representations of the basic types. For example, here's a four byte representations of int data:



00000000 00000000 00000000 00000000 = int 0

00000000 00000000 00000000 00000001 = int 1

00000000 00000000 00000000 00000010 = int 2

00000000 00000000 00000000 00000100 = int 4

00000000 00000000 00000000 00110001 = int 49

00000000 00000000 00000000 01000001 = int 65

00000000 00000000 00000000 11111111 = int 255

BINARY VS. TEXT FILES

In text files, which can be read in notepad++ etc. characters are often stored in smaller 2-byte areas by code number:

00000000 01000001 = code 65 = char "A"

00000000 01100001 = code 97 = char "a"

CHARACTERS

All chars are part of a set of 16 bit+ international characters called Unicode.

These extend the American Standard Code for Information Interchange (ASCII) , which are represented by the ints 0 to 127, and its superset, the 8 bit ISO-Latin 1 character set (0 to 255).

There are some invisible characters used for things like the end of lines.

```
char = chr(8)  # Try 7, as well!
print("hello" + char + "world")
```

The easiest way to use stuff like newline characters is to use escape characters.

```
print("hello\nworld")
```

The opposite to chr is: ascii_value = ord("A")

BINARY VS. TEXT FILES

Note that for an system using 2 byte characters, and 4 byte integers:

00000000 00110001 = code 49 = char "1"

Seems much smaller – it only uses 2 bytes to store the character "1", whereas storing the int 1 takes 4 bytes.

However *each* character takes this, so:

00000000 00110001 = code 49 = char "1"

00000000 00110001 00000000 00110010 = code 49, 50 = char "1" "2"

00000000 00110001 00000000 00110010

00000000 00110111 = code 49, 50, 55 = char "1" "2" "7"

Whereas :

00000000 00000000 00000000 01111111 = int 127

BINARY VS. TEXT FILES

In short, it is much more efficient to store anything with a lot of numbers as binary (not text).

However, as disk space is cheap, networks fast, and it is useful to be able to read data in notepad etc. increasingly people are using text formats like XML.

As we'll see, the filetype determines how we deal with files.

OPEN

```
f = open("anotherfile.txt", xxxx)
```

Where `xxxx` is (from the docs):

Character	Meaning
'r'	open for reading (default)
'w'	open for writing, truncating the file first
'x'	open for exclusive creation, failing if the file already exists
'a'	open for writing, appending to the end of the file if it exists
'b'	binary mode
't'	text mode (default)
'+'	open a disk file for updating (reading and writing)
'U'	universal newlines mode (deprecated)

The default mode is 'r' (open for reading text, synonym of 'rt'). For binary read-write access, the mode 'w+b' opens and truncates the file to 0 bytes. 'r+b' opens the file without truncation.

READING DATA

```
f = open(”some_input_file.txt")
data = []
for line in f:
    parsed_line = str.split(line,",")
    data_line = []
    for word in parsed_line:
        data_line.append(float(word))
    data.append(data_line)
print(data)
f.close()
```

OPEN

Full options:

```
open(file, mode='r', buffering=-1, encoding=None,
errors=None, newline=None, closefd=True, opener=None)
```

buffering: makes the stream of data more consistent, preventing hardware issues interfering with the process. Generally the default works fine but you can control bytes read in for very large files.

encoding: the text file format to use; the default UTF-8 is fine in most cases.

errors: how to handle encoding issues, for example the lack of available representations for non-ASCII characters.

newline: controls the invisible characters written to mark the end of lines.

closefd: whether to remove the file ~link when the file is closed.

opener: option for creating more complicated directory and file opening.

For more info, see: https://docs.python.org/3/library/functions.html#open

WITH

The problem with manually closing the file is that exceptions can skip the close statement.
Better then, to use the following form:

```
with open("data.txt") as f:
      for line in f:
            print(line)
```

The `with` keyword sets up a Context Manager, which temporarily deals with how the code runs. This closes the file automatically when the clause is left.
You can nest withs, or place more than one on a line, which is equivalent to nesting.

```
with A() as a, B() as b:
```

We will do lots exercises on opening/reading/creating files later. Stay tuned!

CONTEXT MANAGERS

Context Managers essentially allow pre- or post- execution code to run in the background (like file.close()).

The associated library can also be used to redirect stdout:

```
with contextlib.redirect_stdout(new_target):
```

For more information, see:

https://docs.python.org/3/library/contextlib.html

READING MULTIPLE FILES

Use fileinput library:

```
import fileinput
a = ["file1.txt", "file2.txt", "file3.txt", "file4.txt"]
b = fileinput.input(a)
for line in b:
    print(b.filename())
    print(line)
b.close()
```

https://docs.python.org/3/library/fileinput.html

EASY PRINT TO FILE

```
print(*objects, sep='', end='\n', file=sys.stdout, flush=False)
```

Prints objects to a file (or stout), separated by sep and followed by end. Other than objects, everything must be a kwarg as everything else will be written out.

Rather than a filename, file must be a proper file object (or anything with a write(string) function).

Flushing is the forcible writing of data out of a stream. Occasionally data can be stored in a buffer longer than you might like (for example if another program is reading data as you're writing it, data might get missed is it stays a while in memory), flush forces data writing.

FILE TYPES

CSV, XML, HTML, JSON, etc.

CSV

Classic format Comma Separated Variables (CSV).

Easily parsed.

No information added by structure, so an ontology (in this case meaning a structured knowledge framework) must be externally imposed.

We've seen one way to read this.

```
10,10,50,50,10
10,50,50,10,10
25,25,75,75,25
25,75,75,25,25
50,50,100,100,50
50,100,100,50,50
```

```
import csv
f = open('../files/csv_files/csv_plain_file.csv', newline='')
reader = csv.reader(f, quoting=csv.QUOTE_NONNUMERIC)
for row in reader: # A list of rows
    for value in row: # A list of value
        print(value) # Floats
f.close()        # Don't close until you are done with the reader
```

CSV READER

```
import csv
with open('./csv_files/addresses.csv') as csvfile:
    readCSV = csv.reader(csvfile, delimiter=',')
    firstnames, lastnames, streets, citys, states,zipcodes = [], [], [],[], [], []

    for row in readCSV:
        firstname, lastname, street = row[0], row[1], row[2]
        city, state, zipcode  = row[3], row[4], row[5]

        firstnames.append(firstname)
        lastnames.append(lastname)
        zipcodes.append(zipcode)

    print(firstnames)
    print(lastnames)
    print(zipcodes)
```

addresses.csv

John	Doe	120 Jefferson	Riverside	NJ	8075
Jack	McGinnis	220 hobo Av.	Phila	PA	9119
John "Da Man"	Repici	120 Jefferson	Riverside	NJ	8075
Stephen	Tyler	7452 Terrace	SomeTown	SD	91234
	Blankman		SomeTown	SD	298
Joan "the bone", Anne	Jet	9th, at Terra	Desert City	CO	123

CSV.READER

```
import csv
f = open('../files/csv_files/plain_csv_file.csv', newline='')
reader = csv.reader(f, quoting=csv.QUOTE_NONNUMERIC)
for row in reader:                                  # A list of rows
    for value in row:                               # A list of value
        print(value)                                # Floats
f.close()       # Don't close until you are done with the reader;
                # the data is read on request.
```

The kwarg `quoting=csv.QUOTE_NONNUMERIC` converts numbers into floats. Remove to keep the data as strings.

Note that there are different dialects of csv which can be accounted for:
https://docs.python.org/3/library/csv.html

For example, add `dialect='excel-tab'` to the reader to open tab-delimited files.

CSV.WRITER

```
f2 = open('dataout.csv', 'w', newline='')
writer = csv.writer(f2, delimiter=' ')
for row in data:
        writer.writerow(row)        # List of values.
f2.close()
```

The optional delimiter here creates a space delimited file rather than csv.

READ CSV – WRITE CSV EXAMPLE

```
import csv
f2 = open('dataout1.csv', 'w', newline='')
writer = csv.writer(f2, delimiter=',')

with open('../files/csv_files/addresses.csv') as csvfile:
    readCSV = csv.reader(csvfile, delimiter=',')
    firstnames, lastnames, streets, citys, states,zipcodes = [], [], [],[], [], []

    for row in readCSV:
        firstname, lastname, street = row[0], row[1], row[2]
        city, state, zipcode  = row[3], row[4], row[5]
        data2write = (firstname,lastname,state)
        print(data2write)
        writer.writerow(data2write)

    f2.close()
```

READ CSV – WRITE TXT EXAMPLE

```
import csv

with open('../files/csv_files/addresses.csv') as csvfile:
    f2 = open("textout.txt", "w")

    readCSV = csv.reader(csvfile, delimiter=',')
    firstnames, lastnames, streets, citys, states,zipcodes = [], [], [],[], [], []

    for row in readCSV:
        firstname, lastname, street = row[0], row[1], row[2]
        city, state, zipcode  = row[3], row[4], row[5]
        text2write = firstname+' lives in '+ city +', '+state
        print(text2write)
        f2.write(text2write+'\n')

    f2.close()
```

JSON (JAVASCRIPT OBJECT NOTATION)

Designed to capture JavaScript objects.

Increasing popular light-weight data format.

Text attribute and value pairs.

Values can include more complex objects made up of further attribute-value pairs.

Easily parsed.

Small(ish) files.

Limited structuring opportunities.

```
{
 "type": "FeatureCollection",
 "features": [ {
  "type": "Feature",
  "geometry": {
    "type": "Point",
    "coordinates": [42.0, 21.0]
   },
  "properties": {
    "prop0": "value0"
  }
 }]
}
```

GeoJSON example

WRITING JSON TO A FILE

```
import json
data = {}
data['people'] = []
data['people'].append({
    'name': 'Scott', 'website': 'stackabuse.com', 'from': 'Nebraska'
})
data['people'].append({
    'name': 'Larry', 'website': 'google.com', 'from': 'Michigan'
})
data['people'].append({
    'name': 'Tim', 'website': 'apple.com', 'from': 'Alabama'
})
with open('data.txt', 'w') as outfile:
    json.dump(data, outfile)
```

READING JSON FROM A TXT FILE

```
import json
with open('./outputs/json_data.txt') as json_file:
    data = json.load(json_file)
    for p in data['people']:
        print('Name: ' + p['name'])
        print('Website: ' + p['website'])
        print('From: ' + p['from'])
        print('')
```

JSON READ

```
import json


f = open('../files/json_files/geo_data.json')
data = json.load(f)
f.close()
print(data)
print(data["features"])
print(data["features"][0]["geometry"])


for i in data["features"]:
    print(i["geometry"]["coordinates"][0])
```

Numbers are converted to floats etc.

```
{
 "type": "FeatureCollection",
 "features": [ {
  "type": "Feature",
  "geometry": {
    "type": "Point",
    "coordinates": [42.0, 21.0]
   },
  "properties": {
    "prop0": "value0"
  }
 }]
}
```

```
{'features':
[
{'type': 'Feature', 'geometry':
                {'coordinates': [42.0, 21.0], 'type':
'Point'},
                'properties': {'prop0': 'value0'}
}
],
'type': 'FeatureCollection'}
```

CONVERSIONS

JSON	Python
object	dict
array	list
string	str
number (int)	int
number (real)	float
true	True
false	False
null	None

It also understands NaN, Infinity, and -Infinity as their corresponding float values, which is outside the JSON spec.

From: https://docs.python.org/3/library/json.html#encoders-and-decoders

JSON WRITE

```
import json

f = open('data.json')
data = json.load(f)
f.close()

f = open('out.json', 'w')
json.dump(data, f)
f.close()
```

SERIALISATION

Serialisation is the converting of code objects to a storage format; usually some kind of file.

Marshalling in Python is essentially synonymous, though in other languages has slightly different uses (for example, in Java marshalling and object may involve additional storage of generic object templates).

Deserialisation (~unmarshalling): the conversion of storage-format objects back into working code.

The json code essentially does this for simple and container Python variables.

For more complicated objects, see pickle: https://docs.python.org/3/library/pickle.html

FORMATTED PRINTING

json.loads and json.dumps convert Python objects to JSON strings.Dumps has a nice print formatting options:

```
print(json.dumps(data["features"], sort_keys=True, indent=4))
[
    {
        "geometry": {
            "coordinates": [
                42.0,
                21.0
            ],
            "type": "Point"
        },
        "properties": {
            "prop0": "value0"
        },
        "type": "Feature"
    }
]
```

More on the JSON library at: https://docs.python.org/3/library/json.html

JSON CHECKING TOOL

```
python -m json.tool < data.json
```

Will print the JSON if right, or suggest corrections.

There are so many Python packages which simplify working with any type of JSON data. We'll use them, if needed.
Let's move on with other file-types.

MARKUP LANGUAGES

Tags and content.

Tags often note the ontological context of the data, making the value have meaning: that is determining its semantic content.

All based on Standard Generalized Markup Language (SGML) [ISO 8879]

HTML
HYPERTEXT MARKUP LANGUAGE

Nested tags giving information about the content.

```
<HTML>
        <BODY>
                <P><B>This</B> is<BR>text  </BODY>
</HTML>
```

Note that tags can be on their own, some by default, some through sloppiness.

Not case sensitive.

Contains style information (though use discouraged).

XML
EXTENSIBLE MARKUP LANGUAGE

More generic.

Extensible – not fixed terms, but terms you can add to.

Vast number of different versions for different kinds of information.

Used a lot now because of the advantages of using human-readable data formats. Data transfer fast, memory cheap, and it is therefore now feasible.

GML

Major geographical type is GML (Geographical Markup Language).

Given a significant boost by the shift of Ordnance Survey from their own binary data format to this.

Controlled by the Open GIS Consortium:

http://www.opengeospatial.org/standards/gml

```
<gml:Point gml:id="p21“
           srsName="http://www.opengis.net/def/crs/EPSG/0/4326">

  <gml:coordinates>45.67, 88.56</gml:coordinates>

</gml:Point>
```

HTML / XML

The two most useful standard libraries are:

Markup: for processing HTML/XML:

https://docs.python.org/3/library/markup.html

And Internet, for gain

https://docs.python.org/3/library/internet.html

http://docs.python-requests.org/en/master/

OS AND PATH

os

The os module allows interaction with the Operating System, either generically or specific to a particular OS.

https://docs.python.org/3/library/os.html

Including:

Environment variable manipulation.

File system navigation.

ENVIRONMENT VARIABLES

These are variables at the OS level, for the whole system and specific users.

For example, include the PATH to look for programs.

```
os.environ
```

A mapping object containing environment information.

```
import os
print(os.environ["PATH"])
print(os.environ["HOME"])
```

For more info on setting Env Variables, see:

https://docs.python.org/3/library/os.html#os.environ

OS FUNCTIONS

```
os.getcwd()                    # Current working directory.
os.chdir('/temp/')             # Change cwd.
os.listdir(path='.')           # List of everything in the present directory.
os.system('mkdir test')        # Run the command mkdir in the system shell
```

OS WALK

A useful method for getting a whole directory structure and files is os.walk.

Here we use this to delete files:

```
for root, dirs, files in os.walk(deletePath, topdown=False):
            for name in dirs:
                  os.rmdir(os.path.join(root, name))
            for name in files:
                  os.remove(os.path.join(root, name))
```

DON'T COPY/PASTE/RUN THIS CODE. IT WILL DELETE THE ENTIRE DIRECTORY

PATHLIB

A library for dealing with file paths:

https://docs.python.org/3/library/pathlib.html

Path classes are either “Pure”: abstract paths not attached to a real filesystem (they talk of path “flavours”); or “Concrete” (usually without “Pure” in the name): attached to a real filesystem. In most cases the distinction is not especially important as the main functions are found in both.

CONSTRUCTING PATHS (FOR WINDOWS USERS)

```
p = pathlib.Path('c:/Program Files') / 'Notepad++'
```

Uses forward slash operators outside of strings to combine paths "/".

Though more platform independent is:

```
a = os.path.join(pathlib.Path.cwd().anchor, 'Program Files', 'Notepad++')
#See next slides for detail.
p = pathlib.Path(a)
str(p)
```

Out: `c:\Program Files\Notepad++`

```
repr(p)
```

Out: `WindowsPath('C:/Program Files/Notepad++')`

For other ways of constructing paths, see:
https://docs.python.org/3/library/pathlib.html#pure-paths

PATH VALUES

`p.name`	final path component.
`p.stem`	final path component without suffix.
`p.suffix`	suffix.
`p.as_posix()`	string representation with forward slashes (/):
`p.resolve()`	resolves symbolic links and “..”
`p.as_uri()`	path as a file URI: file://a/b/c.txt
`p.parts`	a tuple of path components.
`p.drive`	Windows drive from path.
`p.root`	root of directory structure.

PATH VALUES

`pathlib.Path.cwd()`	current working directory.
`pathlib.Path.home()`	User home directory
`p.anchor`	drive + root.
`p.parents`	immutable sequence of parent directories:

```
p = PureWindowsPath('c:/a/b/c.txt')
p.parents[0]    # PureWindowsPath('c:/a/b')
p.parents[1]    # PureWindowsPath('c:/a')
p.parents[2]    # PureWindowsPath('c:/')
```

PATH PROPERTIES

`p.is_absolute()`	Checks whether the path is not relative.
`p.exists()`	Does a file or directory exist.
`os.path.abspath(path)`	Absolute version of a relative path.
`os.path.commonpath(paths)`	Longest common sub-path.
`p.stat()`	Info about path (.st_size; .st_mtime)

https://docs.python.org/3/library/pathlib.html#pathlib.Path.stat

`p.is_dir()`	True if directory.
`p.is_file()`	True if file.
`p.read()`	A variety of methods for reading files as an entire object, rather than parsing it.

PATH PROPERTIES

Listing subdirectories:

```
import pathlib


p = pathlib.Path('.')
for x in p.iterdir():
      if x.is_dir():
            print(x)
```

PATH MANIPULATION

Rename top file or directory to `target`.

```
p.rename(target)
```

Returns new path with changed filename.

```
p.with_name(name)
```

Returns new path with the file extension changed.

```
p.with_suffix(suffix)
```

Remove directory; must be empty.

```
p.rmdir()
```

PATH/FILE MANIPULATION

```
# "Touch" file; i.e. make empty file.
p.touch(mode=0o666, exist_ok=True)


# Make directory.
p.mkdir(mode=0o666, parents=False, exist_ok=False)
  # If parents=True any missing parent directories will be created.
  # exist_ok controls error raising.
```

PATH/FILE MANIPULATION

```
import os
!touch junk1.dat junk2.dat
os.remove('junk1.dat') # Delete file
os.rename('junk2.dat', 'junk2b.dat') # Rename file
os.mkdir('mydata') # Making new directory

# Path functions:
os.path.exists('/tmp') # Checks if loc exists
# Splits loc into directory and file
os.path.split('/usr/local')
# Splits loc into path+file and extension
os.path.splitext('/var/log/messages.gz')
```

PATH MANIPULATION

To set file permissions and ownership, see:

https://docs.python.org/3/library/os.html#os.chmod

https://docs.python.org/3/library/os.html#os.chown

The numbers in `0o666`, the mode above, are left-to-right the owner, group, and public permissions, which are fixed by base 8 (octal) numbers (as shown by "`0o`"). Each is a sum of the following numbers:

```
4 = read

2 = write

1 = execute
```

So here all three are set to read and write permissions, but not execute. You'll see you can only derive a number from a unique set of combinations. This is the classic POSIX file permission system. The Windows one is more sophisticated, which means Python interacts with it poorly, largely only to set read permission.

GLOB

Glob is a library for pattern hunting in files and directories:

https://docs.python.org/3/library/glob.html

```
import glob

# Getting list of all files in current directory:
filelist1 = glob.glob('*') # or
filelist1 = glob.glob('./*')

# Getting list of all files with the extension
‘.py’:
filelist2 = glob.glob('.py')
```

GLOB

```
import glob

glob.glob('**/*.txt', recursive=True)
```

The “**” pattern makes it recursively check the path directory and all subdirectories. With large directory structures this can take a while.

SOME OTHER I/O LIBRARIES

tempfile — Generate temporary files and directories:

https://docs.python.org/3/library/tempfile.html

shutil — High-level file operations, like copying files and directory structures:

https://docs.python.org/3/library/shutil.html

ACCESSING ONLINE FILES

```
import requests
import csv
CSV_URL = 'http://samplecsvs.s3.amazonaws.com/Sacramentorealestatetransactions.csv'
with requests.Session() as s:
    download = s.get(CSV_URL)
    decoded_content = download.content.decode('utf-8')
    cr = csv.reader(decoded_content.splitlines(), delimiter=',')
    my_list = list(cr)
    for row in my_list:
        print(row)
```

ACCESSING ONLINE FILES

```
import requests
import csv
url = 'http://climatedataapi.worldbank.org/climateweb/rest/v1/country/cru/tas/year/CAN.csv'
response = requests.get(url)
if response.status_code != 200:
    print('Failed to get data:', response.status_code)
else:
     wrapper = csv.reader(response.text.strip().split('\n'))
     results = []
     for record in wrapper:
         if record[0] != 'year':
             year = int(record[0])
             value = float(record[1])
             print(year, value)
```

ACCESSING WEBSITES

```
import requests

import csv

response = requests.get('http://datascience.umbc.edu/')

print(response.text)
```

WEB CRAWLING

```
from urllib.request import urlopen
import re
# conntect to a url
website = urlopen('http://datascience.umbc.edu/')
# read htmlcode
html = website.read().decode('utf-8')

#use re.findall to get all the links
links = re.findall('"((http|ftp)s?://.*?)"', html)

# print links
for onelink in links:
    print(onelink)
```

EMAIL CRAWLING

```
website = urlopen('https://dps.umbc.edu/staffdirectory')

# read htmlcode

html = website.read().decode('utf-8')


#use re.findall to get all the links

email_addresses = re.findall(r"[a-z0-9\.\-+_]+@[a-z0-9\.\-+_]+\.[a-z]+", html)


# print email_addresses

for an_email_address in email_addresses:

    print(an_email_address)
```

NOW WE CAN SEND SPAM EMAILS TO THESE ADDRESSES :)

QUESTIONS?

We need coffee!