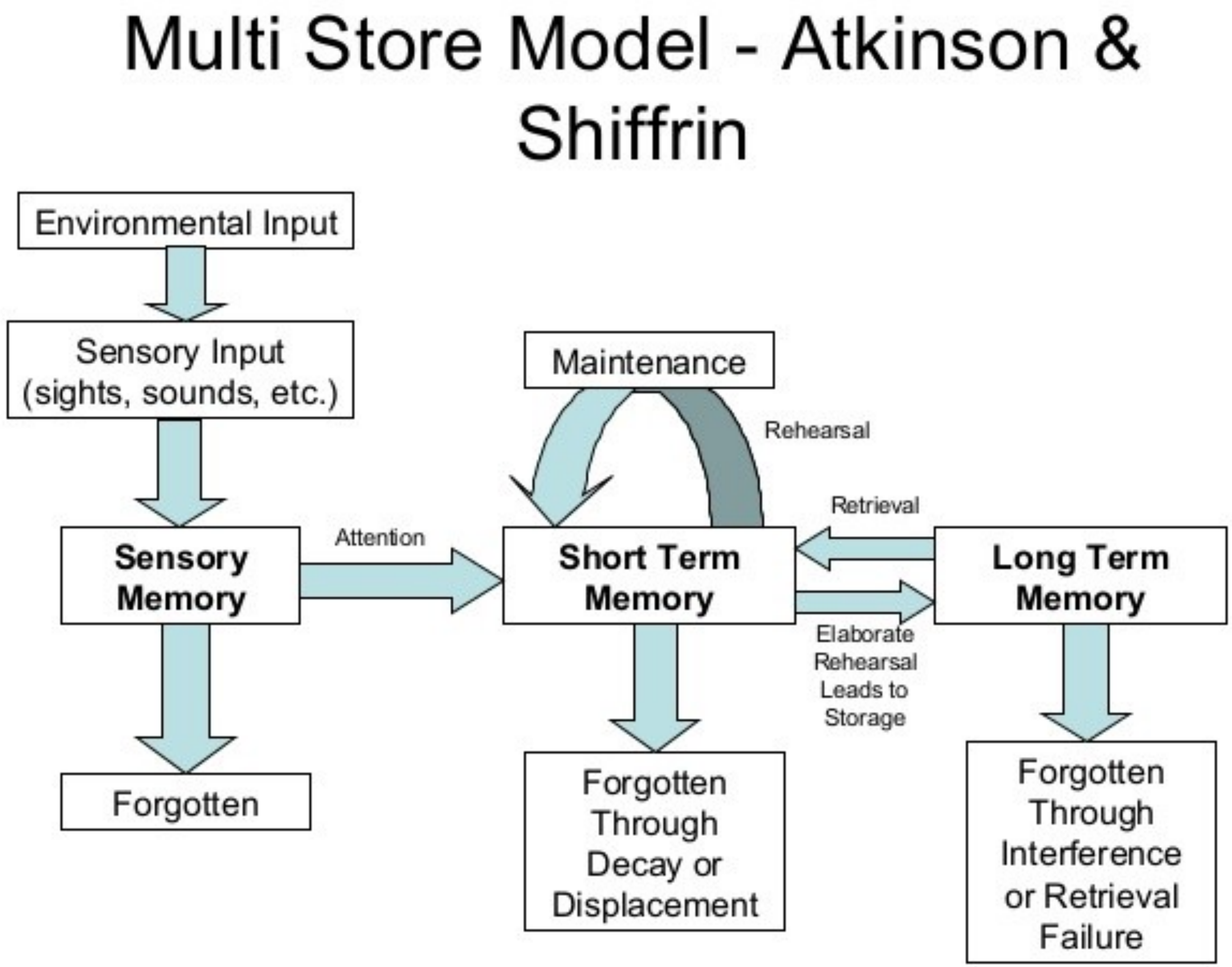
Files and File Operations

CS111 Computer Programming

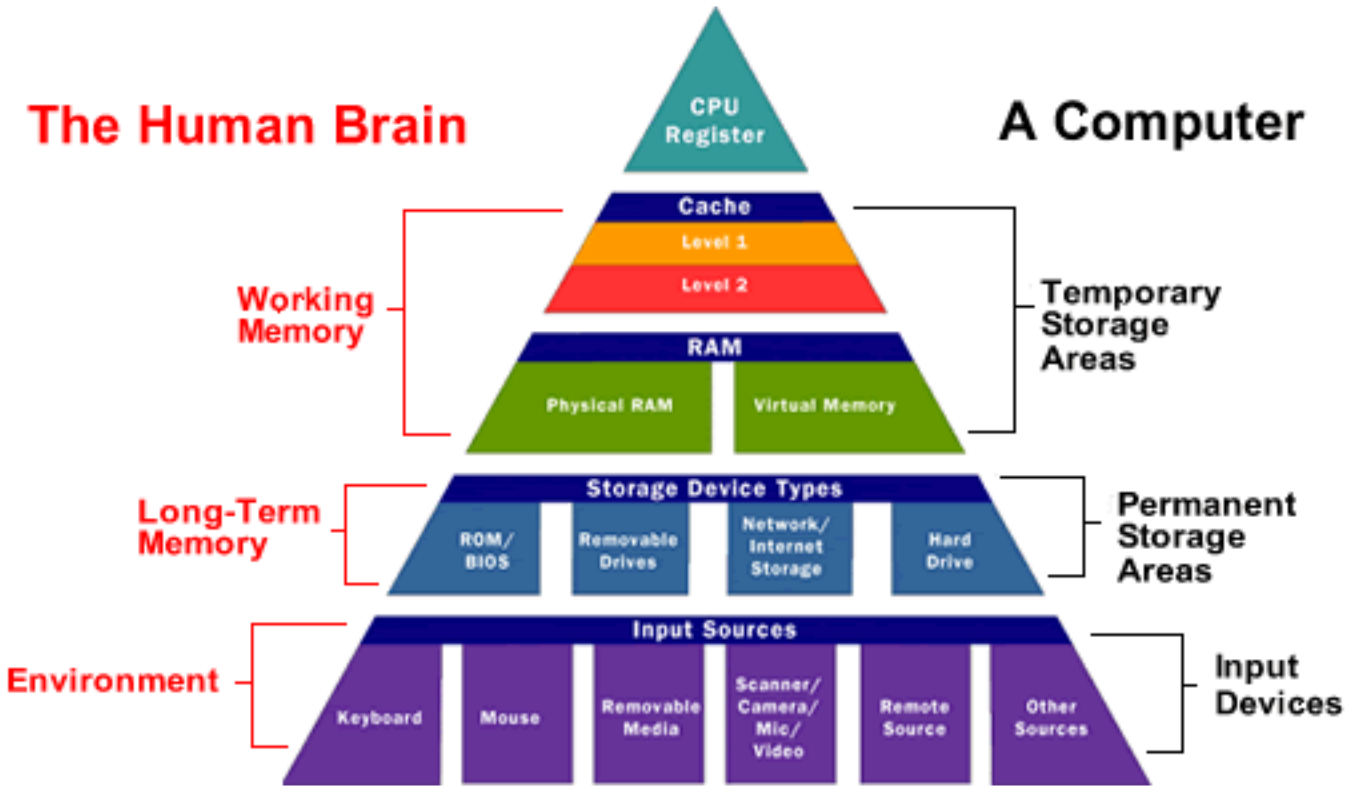
Department of Computer Science
Wellesley College

Computers as a Model for the Human Brain



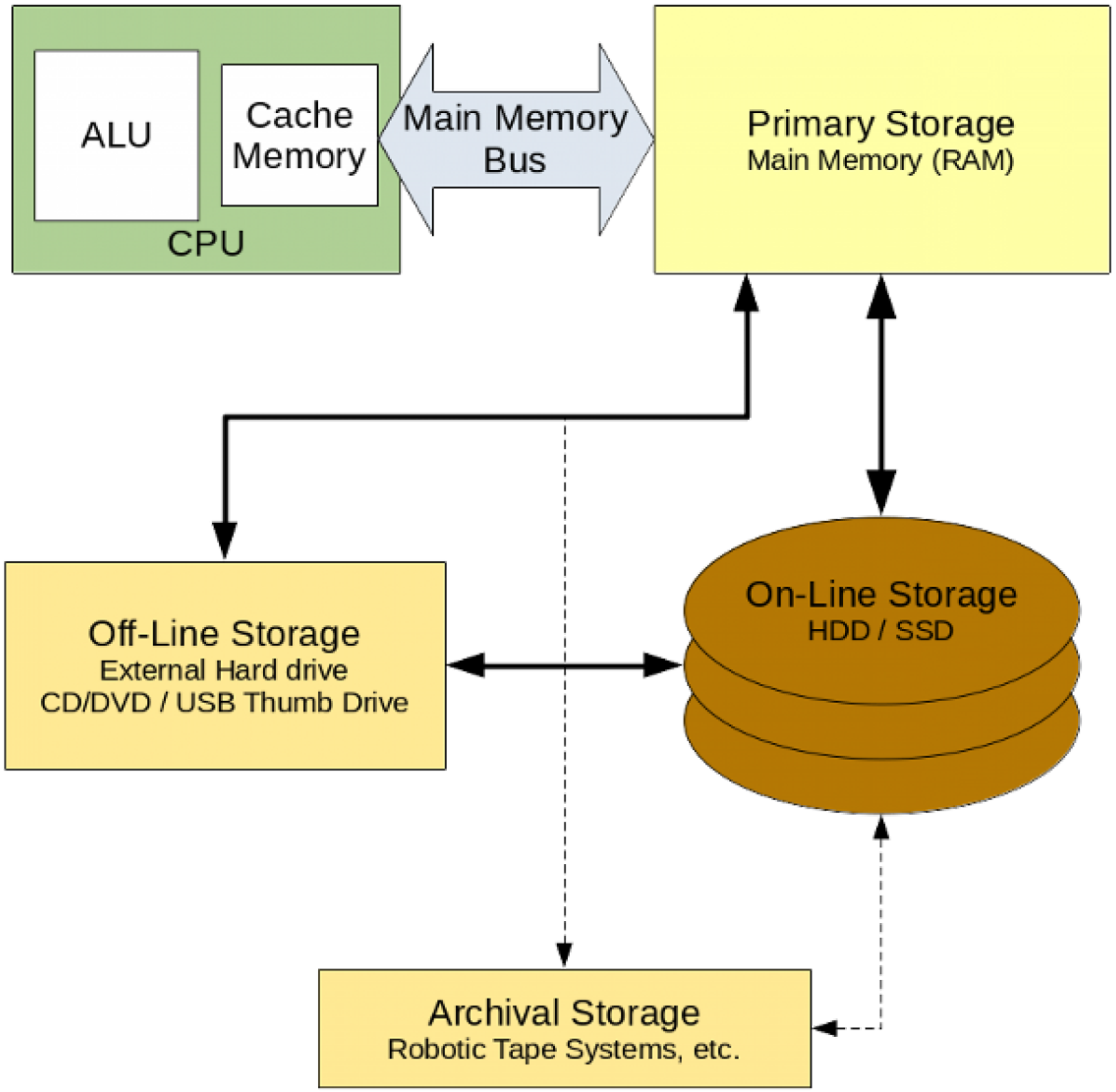
https://en.wikipedia.org/wiki/Information_processing_theory

The Analogy: Human Brain & Computer



https://medium.com/designerd/creative-vs-critical-thinking-2d10d28b0f6c

Computer Memory and Storage Model



https://www.redhat.com/sysadmin/memory-and-storage

Concepts in this slide: Persistent vs. volatile memory. The bit as the unit of information.

Variables vs Files

Variables "reside" in the memory, they exist only for the duration of the program execution. Variables refer to memory locations where values are stored.

Files "reside" in the hard drive (or some external storage). They record data in a persistent way, data that will exist beyond the execution of a particular computer program. Files have a name and an extension.

A computer has two kinds of storage: volatile memory (i.e., the RAM) and persistent memory (i.e, the hard disk). The RAM memory is faster, but more expensive, so current computers have 8-16 GB of it, while a hard disk is slower, but cheaper, so current computers have up to 1-2 TB.

Variables vs Files

Concepts in this slide: Persistent vs. volatile memory. The bit as the unit of information.

The amount of information stored in a file is measured in bytes, kilobytes, megabytes, etc. Various symbols that represent content are internally expressed in bits using standards such as ASCII, Unicode, etc.

Measuring information

The unit of information is the bit. Since one character can be represented in 8 bits (or one byte), byte has become the unit for measuring information stored in a file.

1 B (byte) = 8 bits

1 KB = 1,000 B

1 MB = 1,000,000 B

1 GB = 1,000,000,000 B

1 TB = 1,000,000,000,000 B

Reminder: ASCII Table

Decimal - Binary - Octal - Hex – ASCII Conversion Chart

Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII
0	00000000	000	00	NUL	32	00100000	040	20	SP	64	01000000	100	40	@	96	01100000	140	60	`
1	00000001	001	01	SOH	33	00100001	041	21	!	65	01000001	101	41	A	97	01100001	141	61	a
2	00000010	002	02	STX	34	00100010	042	22	“	66	01000010	102	42	B	98	01100010	142	62	b
3	00000011	003	03	ETX	35	00100011	043	23	#	67	01000011	103	43	C	99	01100011	143	63	c
4	00000100	004	04	EOT	36	00100100	044	24	$	68	01000100	104	44	D	100	01100100	144	64	d
5	00000101	005	05	ENQ	37	00100101	045	25	%	69	01000101	105	45	E	101	01100101	145	65	e
6	00000110	006	06	ACK	38	00100110	046	26	&	70	01000110	106	46	F	102	01100110	146	66	f
7	00000111	007	07	BEL	39	00100111	047	27	‘	71	01000111	107	47	G	103	01100111	147	67	g
8	00001000	010	08	BS	40	00101000	050	28	(	72	01001000	110	48	H	104	01101000	150	68	h
9	00001001	011	09	HT	41	00101001	051	29	)	73	01001001	111	49	I	105	01101001	151	69	i
10	00001010	012	0A	LF	42	00101010	052	2A	*	74	01001010	112	4A	J	106	01101010	152	6A	j
11	00001011	013	0B	VT	43	00101011	053	2B	+	75	01001011	113	4B	K	107	01101011	153	6B	k
12	00001100	014	0C	FF	44	00101100	054	2C	,	76	01001100	114	4C	L	108	01101100	154	6C	l
13	00001101	015	0D	CR	45	00101101	055	2D	-	77	01001101	115	4D	M	109	01101101	155	6D	m
14	00001110	016	0E	SO	46	00101110	056	2E	.	78	01001110	116	4E	N	110	01101110	156	6E	n
15	00001111	017	0F	SI	47	00101111	057	2F	/	79	01001111	117	4F	O	111	01101111	157	6F	o
16	00010000	020	10	DLE	48	00110000	060	30	0	80	01010000	120	50	P	112	01110000	160	70	p
17	00010001	021	11	DC1	49	00110001	061	31	1	81	01010001	121	51	Q	113	01110001	161	71	q
18	00010010	022	12	DC2	50	00110010	062	32	2	82	01010010	122	52	R	114	01110010	162	72	r
19	00010011	023	13	DC3	51	00110011	063	33	3	83	01010011	123	53	S	115	01110011	163	73	s
20	00010100	024	14	DC4	52	00110100	064	34	4	84	01010100	124	54	T	116	01110100	164	74	t
21	00010101	025	15	NAK	53	00110101	065	35	5	85	01010101	125	55	U	117	01110101	165	75	u
22	00010110	026	16	SYN	54	00110110	066	36	6	86	01010110	126	56	V	118	01110110	166	76	v
23	00010111	027	17	ETB	55	00110111	067	37	7	87	01010111	127	57	W	119	01110111	167	77	w
24	00011000	030	18	CAN	56	00111000	070	38	8	88	01011000	130	58	X	120	01111000	170	78	x
25	00011001	031	19	EM	57	00111001	071	39	9	89	01011001	131	59	Y	121	01111001	171	79	y
26	00011010	032	1A	SUB	58	00111010	072	3A	:	90	01011010	132	5A	Z	122	01111010	172	7A	z
27	00011011	033	1B	ESC	59	00111011	073	3B	;	91	01011011	133	5B	[	123	01111011	173	7B	{
28	00011100	034	1C	FS	60	00111100	074	3C	<	92	01011100	134	5C	\	124	01111100	174	7C	|
29	00011101	035	1D	GS	61	00111101	075	3D	=	93	01011101	135	5D	]	125	01111101	175	7D	}
30	00011110	036	1E	RS	62	00111110	076	3E	>	94	01011110	136	5E	^	126	01111110	176	7E	~
31	00011111	037	1F	US	63	00111111	077	3F	?	95	01011111	137	5F	_	127	01111111	177	7F	DEL

There are lots of file extensions

https://www.howtogeek.com/356448/what-is-a-file-extension/

File Extensions

By convention, **file extensions (e.g., .pdf)** indicate the content of a file. Computer programs use other means to detect file content. Here is a list of files that we have used or will be using:

.txt – plain text file, readable with a text editor. Lines separated by '\n'.

.csv – comma separated values. Simple text, but can be read by spreadsheet applications as well, e.g., Microsoft Excel or Google Sheets.

.json – **J**ava**S**cript **O**bject **N**otation (JSON): data structured in lists, dictionaries, or a combination of thereof. Can be viewed as text, and loaded directly into Python with the module `json`.

.py – A text file in which we store Python programs. When a program like Thonny opens a .py file, it highlights text according to Python syntax.

.ipynb – interactive python notebook. A JSON file that the Jupyter notebook program interprets as a web page.

Working with files in Python

Concepts in this slide:
The built-in function `open` to create file objects.

Until now, our data have been stored in variables. However, the universal way of storing data is in a file, which is persistent storage and can be used across applications.

In Python, `open` creates and returns a file object

```
In [ ]: myFile = open('thesis.txt', 'r')
In [ ]: print(myFile)
<_io.TextIOWrapper name = 'thesis.txt' mode='r'
encodings='UTF-8'>
In [ ]: type(myFile)
Out[ ]: _io.TextIOWrapper
```

_io.TextIOWrapper is a class that defines a file object that interprets the files as a stream of text.

Working with files in Python

Concepts in this slide:
The built-in function open to create file objects.

```
In [ ]: myFile = open('thesis.txt', 'r')
In [ ]: print(myFile)
<_io.TextIOWrapper name = 'thesis.txt' mode='r'
encodings='UTF-8'>
In [ ]: type(myFile)
Out[ ]: _io.TextIOWrapper
```

A file can be opened for reading (`'r'`), writing (`'w'`), or appending (`'a'`).

We **cannot** write or append in a file opened for reading.

Working with files in Python

Concepts in this slide:
The built-in function `open` to create file objects.

```
In [ ]: myFile = open('thesis.txt', 'r')
In [ ]: print(myFile)
<_io.TextIOWrapper name = 'thesis.txt' mode='r'
encodings='UTF-8'>
In [ ]: type(myFile)
Out[ ]: _io.TextIOWrapper
```

The most important methods for a file object are: **`read`**, **`readlines`**, **`readline`**, and **`write`**, that we'll cover today.

Context: Python Objects

In Python, every value is an object, an instance of a particular class. A class is a special construct that defines a new type. When we call the function **type** with a value, it tells us the name of its class. You will learn about classes in CS 230.

Objects have attributes and methods. We have been using them throughout the semester.

A file object

```
<_io.TextIOWrapper
name = 'thesis.txt'
mode='r'
encodings='UTF-8'>
```

_io.TextIOWrapper	
name: "thesis.txt"	Attributes
mode: 'r'	
encoding: 'UTF-8'	
read()	Methods
readline()	
readlines()	

Image credit: https://www.analyticsvidhya.com/blog/2021/05/oop-in-python-for-absolute-beginners/

Context: Python Method Calls

We've already seen many examples of calling methods on Python objects in the context of strings and lists. For example:

Method Calls	Output Values
`'hello'.upper()`	`'HELLO'`
`'Hello'.lower()`	`'hello'`
`'bat or cat or dog'.split()`	`['bat','or','cat','or','dog']`
`'bat or cat or dog'.split(' or ')`	`['bat','cat','dog']`
`'; '.join(['bat','cat','dog'])`	`'bat; cat; dog'`
`L = [8,4,5]`	
`L.append(7)`	`None`
`L.insert(2,9)`	`None`
`L.pop(0)`	`8`

Method calls are similar to function calls in the sense that both have comma-separated arguments that appear in parens. However, in a method call, the object on which the method is called (the **receiver**) appears **before** the method name, separated from it by a dot.

Preferred file opening syntax: `with … as`

Concepts in this slide:
How to open files with the notation `with … as`

It's easy to forget to close a file. This usually isn't too bad when **reading** a file, but can be disastrous when **writing** a file (the contents may not actually be written until the file is closed!)

Python's `with … as` notation for files implicitly closes a file, even if an error occurs within the file operations.

`'r'` – read mode; `'w'` – write mode

```
f = open(filename, 'r')
… file operations involving f …
f.close()
```

⟺

```
with open(filename, 'r') as f:
    … file operations involving f …
    # f implicitly closed
    # when with is done.
```

```
f = open(filename, 'w')
… file operations involving f …
f.close()
```

⟺

```
with open(filename, 'w') as f:
    … file operations involving f …
    # f implicitly closed
    # when with is done.
```

Reading all text with read

cities.txt

```
Wilmington
Philadelphia
Boston
Charlotte
```

```
# reads all lines at once as string
with open('cities.txt', 'r') as inputFile:
    allText = inputFile.read()

In []: allText
Out[]: 'Wilmington\nPhiladelphia\nBoston\nCharlotte'

In []: allText.split()
Out[]: ['Wilmington', 'Philadelphia', 'Boston', 'Charlotte']

In []: moreText = inputFile.read()
```

ValueError: I/O operation on closed file

To notice:
The method **read** returns all the content of the file as a single string. Notice how the returned string contains the newline characters. These are chars that allow the two other methods **readline** and **readlines** to know how to recognize lines.
In this case too, if we try to access **inputFile** outside the **with** … **as** block, we'll get an error about operations with a closed file.

Reading <u>all lines</u> with **readlines**

```
# reads all lines at once as list of strings
with open('cities.txt', 'r') as inputFile:
    allLines = inputFile.readlines()
```

```
In []: allLines
Out[]: ['Wilmington\n', 'Philadelphia\n',
        'Boston\n', 'Charlotte\n']
```

cities.txt

```
Wilmington
Philadelphia
Boston
Charlotte
```

To notice:

The method **readlines** returns a list of all lines in a file. Each line is a string terminated by the newline character `'\n'`. These newline characters are visible in the Out[] cell, but not if we print each line. They are also not visible in the text file as well (see green box).

Important: Use **readlines** only when a file is not too large. This is because it will read all content and store it in the RAM memory (which, as you read in slide 2, is limited).

Reading one line with readline

```
# reads one line at a time
with open('cities.txt', 'r') as inputFile:
    line1 = inputFile.readline()
    line2 = inputFile.readline()

In []: line1
Out[]: 'Wilmington\n'

In []: line2
Out[]: 'Philadelphia\n'

In []: line3 = inputFile.readline()
```

ValueError: I/O operation on closed file

cities.txt

```
Wilmington
Philadelphia
Boston
Charlotte
```

To notice:

Interpret the first line as saying: open the file 'cities.txt' for reading and assign the variable **inputFile** to it, so that we can access its content.

The indented statement is calling the method **readline** on the file object **inputFile** to read only the first line and save it in the variable **oneline**. Afterwards, the file is closed, which we can notice if we try to use **readline** again on the **inputFile**.

Reading line-by-line with a for loop

Concepts in this slide:
Within a for loop, there is no need to explicitly call the three read methods.

```
def linesFromFile(filename):
    '''Returns a list of all lines in the given file. In
    each line, the terminating newline has been removed.
    '''
    with open(filename, 'r') as inputFile: # open the file
        strippedLines = []
        for line in inputFile:
            strippedLines.append(line.strip())
        return strippedLines
```

No explicit method with the file object. That is, no `read`, `readlines`, or `readline`.

To notice:
Within a **`for`** loop to read the content of a file, we don't need to call explicitly any of the three methods that we saw. A file object is an iterator, it knows how to iterate over its elements, which are the lines denoted by the newline character.
This is our preferred method for reading from a file.
The string method **`strip`** removes the newline character and all white space around the line.

Writing Files

Concepts in this slide: Writing to a file that is opened for writing.

A file can be created (or opened) for writing by providing the argument 'w' to **open**, signifying **write mode**.

When writing files, the syntax **with … as** is very important, because forgetting to close a file has consequences.

```
with open('memories.txt', 'w') as memfileW:
    memfileW.write('get coffee\n')
    memfileW.write('do CS111 homework\n')
    memfileW.write('vote in midterm elections!\n')
```

At this point, the file named memories.txt is stored persistently in the file system with the following contents:

```
get coffee
do CS111 homework
vote in midterm elections!
```

To notice:

- The second argument **'w'** is what opens the file in writing mode.
- The strings to write in the file contain the newline character **'\n'** as their last character to denote that a new line should start in the file.
- The file memFileW is closed automatically.

Appending to files

How do we add lines to an existing file? We can't open the file in **write mode** (with a `'w'`), because that erases all previous contents and starts with an empty file. Instead, we open the file in **append mode** (with an `'a'`). Any subsequent writes are made after the existing contents:

```
with open('memories.txt', 'a') as memfileA:
    memfileA.write('win Nobel prize\n')
    memfileA.write('eat big sundae\n')
```

Now the file memories.txt has the contents:

```
get coffee
do CS111 homework
vote in midterm elections!
win Nobel prize
eat big sundae
```

If a file does not already exist, opening it in **append mode** creates an empty file.

Exception Handling with `try/except`

Concepts in this slide:
New Python keywords: `try` and `except` to catch exceptions.

Misspelled filename:

```
memories = linesFromFile('memory.txt')
```

IOError: No such file or directory: 'memory.txt'

An exception is an error detected during execution of a program.

When part of a program raises an exception (e.g., code for reading a file), it is often better to catch and handle the exception rather than have the program terminate abruptly with an error message (e.g., if a file doesn't exist).

In Python, exceptions can be caught and handled with **`try/except`** statements.

```
try:
    # Lines of code that may raise an exception
except ErrorType:
    # Lines to execute when exception
    # with type ErrorType is raised
```

Exception Handling Examples

```
try:
    memories = linesFromFile('memory.txt')
except IOError:
    memories = []  # Use empty list in this case
```

To remember

We can use try/except whenever we anticipate errors coming from sources that are related to human input (humans often make mistakes).

```
a = 0
try:
    x = 8/a
    print(x)
except ZeroDivisionError:
    print('Do not divide by zero.')
```

```
while True:
    try:
        i = int(raw_input('Please enter an integer: '))
        print('Good, you entered', i)
        break # Python keyword to exit a loop
    except ValueError:
        print('Not a valid integer. Try again... ')
```

Challenge Problem: Reading / Writing Files

Given input: CSV file with information from the Nobel Prize Committee

	A	B	C	D	E	F	G	H
1	**Year**	**Name**		**Gender**	**Citizenship**	**Second Citizenship**	**Born**	**Remarks**
59	1966	Samuel	Agnon	Male	Israel		1888	
60	1966	Nelly	Sachs	Female	Sweden		1891	
61	1965	Mikhail	Sholokhov	Male	Soviet Union		1905	
62	1964	Jean-Paul	Sartre	Male	France		1905	Declined the prize
63	1963	Giorgos	Seferis	Male	Greece		1900	
64	1962	John	Steinbeck	Male	United States		1902	
65	1961	Ivo	Andric	Male	Yugoslavia		1892	
66	1960	Saint-John	Perse	Male	France		1887	
67	1959	Salvatore	Quasimodo	Male	Italy		1901	
68	1958	Boris	Pasternak	Male	Soviet Union		1890	Forced to decline

Desired output: CSV file with total number of prizes per country of citizenship

	A	B
1	**Country**	**Total Wins**
2	France	15
3	United States	12
4	United Kingdom	11
5	Sweden	8
6	Germany	8
7	Italy	6
8	Poland	5
9	Spain	5
10	Ireland	4

This is a real-world accumulation problem and now you have the skills to solve such a problem with Python.
Find all the details in the Jupyter notebook.

Test your knowledge

1. In your own words, what is the difference between a variable and a file?
2. We listed many file extensions in Slide 8. Do you recognize all of them? Have you tried to open any of these files with applications different from the ones which created them (or that you usually open them)? What have you seen?
3. When developing Python programs, why would we need to read files? Why would we need to write files?
4. What would be some good uses for the file methods **`readline`**, **`readlines`**, and **`read`**?
5. What do the three letters **`'r'`**, **`'w'`**, **`'a'`** mean? Do you think one of them can be omitted when opening a file?
6. What would happen if we try to write into a file open for reading?
7. How does the method **`readlines`** recognize lines, so that it can return a list of all lines?
8. **Challenge**: suppose we have a text file open for reading, **`inputFile`**. Can you write one line of code to print the total number of words in the file?