PYTHON ANALYSIS: 1-D (TIME SERIES)

1. LOADING STORED DATA:

Data can be stored in a variety of file formats (text, binary, compressed, etc.) which are normally, but not always, identified by the filename suffix (.txt, .raw, .zip, etc.). Determining how data is stored, and then loading it correctly into RAM (system memory) for subsequent analysis, is often a major obstacle to "getting started" with data processing. Here we will demonstrate how to load two of the standard data file formats (a delimited text file and a raw binary file) into Python for subsequent analysis with NumPy.

TASK 1.1: LOADING A TEXT FILE

Text can be stored in many different representations within a file. However, by the far the most common encoding is ASCII (Figure 1.1). Using the ASCII table, the decimal (and thus 8-bit binary value) for any of 256 different "characters" can be encoded (and decoded). Therefore, in order to save numerical values as text, one must encode every digit and symbol as a "character". For example, the number -7.436 would be saved as the letters '-', '7', '.', '4','3', and '6'. This can require more space, 1-Byte per digit/symbol, but is convenient and readable for small files.

Figure 1.1: *ASCII Look-Up Table*

0	<NUL>	32	<SPC>	64	@	96	`	128	Ä	160	†	192	¿	224	‡
1	<SOH>	33	!	65	A	97	a	129	Å	161	°	193	¡	225	·
2	<STX>	34	"	66	B	98	b	130	Ç	162	¢	194	¬	226	‚
3	<ETX>	35	#	67	C	99	c	131	É	163	£	195	√	227	„
4	<EOT>	36	$	68	D	100	d	132	Ñ	164	§	196	ƒ	228	‰
5	<ENQ>	37	%	69	E	101	e	133	Ö	165	•	197	≈	229	Â
6	<ACK>	38	&	70	F	102	f	134	Ü	166	¶	198	Δ	230	Ê
7	<BEL>	39	'	71	G	103	g	135	á	167	ß	199	«	231	Á
8	<BS>	40	(	72	H	104	h	136	à	168	®	200	»	232	Ë
9	<TAB>	41	)	73	I	105	i	137	â	169	©	201	…	233	È
10	<LF>	42	*	74	J	106	j	138	ä	170	™	202		234	Í
11	<VT>	43	+	75	K	107	k	139	ã	171	´	203	À	235	Î
12	<FF>	44	,	76	L	108	l	140	å	172	¨	204	Ã	236	Ï
13	<CR>	45	-	77	M	109	m	141	ç	173	≠	205	Õ	237	Ì
14	<SO>	46	.	78	N	110	n	142	é	174	Æ	206	Œ	238	Ó
15	<SI>	47	/	79	O	111	o	143	è	175	Ø	207	œ	239	Ô
16	<DLE>	48	0	80	P	112	p	144	ê	176	∞	208	–	240	
17	<DC1>	49	1	81	Q	113	q	145	ë	177	±	209	—	241	Ò
18	<DC2>	50	2	82	R	114	r	146	í	178	≤	210	"	242	Ú
19	<DC3>	51	3	83	S	115	s	147	ì	179	≥	211	"	243	Û
20	<DC4>	52	4	84	T	116	t	148	î	180	¥	212	'	244	Ù
21	<NAK>	53	5	85	U	117	u	149	ï	181	µ	213	'	245	ı
22	<SYN	54	6	86	V	118	v	150	ñ	182	∂	214	÷	246	ˆ
23	<ETB>	55	7	87	W	119	w	151	ó	183	Σ	215	◊	247	˜
24	<CAN>	56	8	88	X	120	x	152	ò	184	∏	216	ÿ	248	¯
25	<EM>	57	9	89	Y	121	y	153	ô	185	π	217	Ÿ	249	˘
26	<SUB>	58	:	90	Z	122	z	154	ö	186	∫	218	/	250	˙
27	<ESC>	59	;	91	[	123	{	155	õ	187	ª	219	€	251	°
28	<FS>	60	<	92	\	124	|	156	ú	188	º	220	‹	252	¸
29	<GS>	61	=	93	]	125	}	157	ù	189	Ω	221	›	253	˝
30	<RS>	62	>	94	^	126	~	158	û	190	æ	222	fi	254	˛
31	<US>	63	?	95	_	127	<DEL>	159	ü	191	ø	223	fl	255	ˇ

EXERCISE 1.1.1: WRITE YOUR NAME IN (DECIMAL) ASCII CODES?

Advanced Exercise 1.1.1: Write your name in Hexadecimal and Binary (using ASCII encoding)?

_____

```
import numpy as np
import matplotlib.pyplot as plt

# Specify the filename path (location)
filename = 'your_data.txt

# Load numbers from a text file with comma separated values (CSV)
data = np.loadtxt(filename)

# Plot the data in a graph
plt.plot(data)
```

EXERCISE 1.1.2: USE NUMPY'S *LOADTXT* FUNCTION TO LOAD A ".TXT" FILE INTO A NUMPY ARRAY AND THEN USE MATPLOTLIB TO PLOT THE DATA IN A GRAPH.

- You can use an single column text file, even if it has a different suffix (i.e. ".csv")

  Advanced Exercise 1.1.2: It is not always this straightforward. Load a dataset from the World Bank (http://data.worldbank.org/). How do you deal with the header and labels?

TASK 1.2: LOADING A BINARY FILE

Data is always stored in a binary form, .e.g. 01000100011000010111010001100001. For an ASCII text file, each sequence of 8-bits defines a text character. However, some files use a "raw binary" format. In these cases, the sequence of bits could represent a list of 16-bit integers, where each 2-byte chunk should be decoded as one number, or they may represent 64-bit floating point values, arranged in 8 columns. Unfortunately, there are no conventions and it is important to try and determine how the file is arranged, and what it encodes, before attempting to load a raw binary file.

ASCII Text File:	01000100011000010111010001100001	= "Data"
16-bit Integers:	0100010001100001 0111010001100001	= 17505 29793
32-bit Integer:	01000100011000010111010001100001	= 1147237473
32-bit Float:	01000100011000010111010001100001	= 901.81842041015625

EXERCISE 1.2.1: LOAD THE BINARY "EXAMPLE.RAW" FILE USING NUMPY *FROMFILE* (THE BINARY FILE CONTAINS DATA TAKEN FROM 32 COLUMNS OF (SIGNED) 16-BIT INTEGERS); RESHAPE THE RAW ARRAY ACCORDINGLY AND PLOT ONE COLUMN IN A GRAPH.

Advanced Exercise 1.2.1: Try to sort the events in this file into distinct "sources". ;)

```
import numpy as np
import matplotlib.pyplot as plt

# Specify the filename path (location)
filename = r'example_binary.raw'

# Load numbers from a binary file
data_type = np.float32
raw_data = np.fromfile(filename, dtype=data_type)

# Now we have a list of all values in the file. They must be reshaped into
#  2D array with the appropriate dimensions (rows x cols).
number_elements = np.size(raw_data)
number_cols = 32
number_rows = number_elements/number_cols
data = np.reshape(raw_data, (number_rows, number_cols))

# Plot some of the data in a graph, one column
plt.plot(data[0:100000,2])
```

2. ANALYZING 1-D TIME SERIES:

You now have a list of values, a one dimensional array, loaded into system memory. There are many useful things that you can do with this data: plot it, find the min/max/mean values. We encourage you to play around with this data and the numerous NumPy functions that can process, manipulate, and visualize it. For now, however, we will focus on the task at hand.

TASK 2.1: FIND THE PEAKS (LOCAL MAXIMA)

We often want to determine specific moments in a time series when the value of the data reaches a "peak", i.e. local maxima. This could represent a spike from a neuron, the onset of movement, or a range of other relevant "events". Here we will develop an algorithm to identify these data "peaks".

EXERCISE 2.1.1: DESCRIBE (IN COLLOQUIAL) LANGUAGE A PROCEDURE FOR IDENTIFYING PEAKS IN A TIME VARYING SIGNAL.

Bonus Exercise 2.1.1: Describe common cases in which your procedure might identify erroneous "peaks".

EXERCISE 2.1.2: USE THE SCIPY FUNCTION *ARGRELEXTREMA* , FOUND IN THE *SIGNAL* SUB-LIBRARY, TO FIND THE LOCATIONS (***ARG***UMENTS) OF ***REL***ATIVE MAXIMA (***EXTREMA***). PLOT THESE LOCATIONS USING COLORED MARKS ON TOP OF THE TIME SERIES DATA.

Bonus Exercise 2.1.2: Why is "np.greater_equal" preferred to "np.greater"?

```
import numpy as np
import scipy.signal as signal
import matplotlib.pyplot as plt

# Specify the filename path (location)
filename = r'example_data.csv'

# Load numbers from a text file with comma separated  values (CSV)
data = np.loadtxt(filename)

# Find the relative maxima (comparing the 10-pts on either side: order=10)
maxima_locations = signal.argrelextrema(data, np.greater_equal, order= 10)[0]

# Plot the data in a graph, mark the peaks with red '+' symbols
plt.plot(data)
plt.plot(maxima_locations, data[maxima_locations], 'r+')
```

TASK 2.2: CLEAN UP THE DATA

Depending on the nature of your data, you will likely notice that the *argrelextrema* function returns locations that you would not consider peaks; noisy fluctuations, multiple maxima points with the same value, etc.

EXERCISE 2.2.1: CLEAN UP THE DETECTED "PEAKS" BY REMOVING THOSE APPERING BELOW A THRESHOLD (LIKELY NOISE).

Bonus Exercise 2.2.1: Write a peak_detect function accepting useful parameters (threshold, order, etc.)?

```
# # Append to previous code # #

# User defined threshold value determines which maxima are valid
value_threshold = 400
good_maxima = data[maxima_locations] > value_threshold

# Reset list of maxima to only those deemed "good"
maxima_locations = maxima_locations[good_maxima]

# Plot the data in a graph, mark the peaks with blue 'o' symbols
plt.plot(maxima_locations, data[maxima_locations], 'bo')
```

EXERCISE 2.2.2: FURTHER CLEAN UP THE DETECTED "PEAKS" BY REMOVING THOSE APPERING TOO CLOSE TOGETHER, I.E. TOO SOON AFTER THE PREVIOUS MAXIMA (ALSO LIKELY NOISE).

Bonus Exercise 2.2.1: Extend your peak_detect function use timing parameters (refractory period)?

```
# # Append to previous code # #

# Only accept maxima that occur after a minimum time from the previous maxima
time_since_prev_maxima = np.diff(maxima_locations)

# Since the first maxima is excluded by the "diff" function, use its idx
#  as the time since previous maxima (since start of recording) and insert it
#  at the start of the array of maxima times
time_since_prev_maxima = np.insert(time_since_prev_maxima, 0,
maxima_locations[0])

# Remove maxima that come too soon
time_threshold = 10
good_maxima = time_since_prev_maxima > time_threshold
# This creates a boolean array: True = maxima is OK, False = it comes too soon

# Reset list of maxima to only those deemed "good"
maxima_locations = maxima_locations[good_maxima]

# Plot the data in a graph, mark the peaks with green 'diamond' symbols
plt.plot(maxima_locations, data[maxima_locations], 'gd')
```

TASK 2.3: MEASURE THE INTER-PEAK INTERVALS

Now that you have identified the relevant events in your data you can begin to measure interesting features, i.e. time between each event (interval) and the trajectory of the data surrounding each event (event-triggered-average). Let's do both of these tasks.

EXERCISE 2.3.1: MEASURE THE INTER-EVENT (INTER-PEAK) INTERVALS AND PLOT THEM.

```
# Append to previous code #

# Measure Inter-peak Times
inter_peak_times = np.diff(maxima_locations)

# Create a new figure and plot intervals as blue 'o' symbols
plt.figure()
plt.plot(inter_peak_times, 'bo')
```

EXERCISE 2.3.2: COMPUTE THE EVENT-TRIGGERED-AVERAGE (EXTRACT DATA WINDOW SURROUNDING EACH EVENT AND DISPLAY AVERAGE TRAJECTORY).

```
# Append to previous code #

# Determine the number of events
num_events = np.size(maxima_locations)

# Specify the size of the window (flank to left and right of event)
flank_size = 20
window_size = flank_size * 2 + 1 # Size of window, including the event centre

# Create empty array for all windows
windows = np.zeros((num_events, window_size))

# Declare a for loop that goes through all maxima locations
i = 0
for m in maxima_locations:
    # Fill in data surrounding current event
    windows[i,:] = data[(m-flank_size):(m+1+flank_size)]
    i = i+1

# Plot all windows and the average window (ETA)
plt.figure()
plt.plot(windows.T)
ETA = np.mean(windows, axis=0)
plt.plot(ETA, 'k', linewidth=5)
```

3. BASIC STATISTICS:

So, did anything “significant” appear in your data? Did the events become more consistent over time? Was one set of events more stereotyped relative to another? Time to start thinking about some statistics…

4. USEFUL NUMPY – SCIPY – MATPLOTLIB FUNCTION

The following is a crude, biased, but hopefully useful list of NumPy functions that often come up in (neuro) science data analysis. Please suggest additions/deletions!

CORE CONCEPTS

Text vs. Binary

What is the difference?

ASCII Table

File vs. System Memory (RAM)

Arrays: Size, Indexing, Slicing