Lecture 18: Command line

Part 1: Calling programs from Python

While we haven't explored many command line programs yet, we should be familiar with a few. For example, grep, which allows you to identify lines of a file that match a certain pattern. For example, we can search the WorldDemographics.csv file to filter lines that match a certain pattern. For example, if we want to identify the demographics for Algeria, we can do the following:

```
(base) ➜  ExampleData grep Algeria WorldDemographics.csv
202,Algeria,12,Country,912,0,1008303
203,Algeria,12,Country,912,1,1008303
204,Algeria,12,Country,912,2,1008303
…
300,Algeria,12,Country,912,98,1493
301,Algeria,12,Country,912,99,1228
302,Algeria,12,Country,912,100,963
```

Now let's say we wanted to run this in a Python script. We can do that very easily using the subprocess module. Subprocess contains a number of useful functions for running commands and spawning new processes. We will cover two of these in this class.

subprocess.**run**(*args*, ***, *stdin=None*, *input=None*, *stdout=None*, *stderr=None*, *capture_output=False*, *shell=False*, *cwd=None*, *timeout=None*, *check=False*, *encoding=None*, *errors=None*, *text=None*, *env=None*, *universal_newlines=None*, ***other_popen_kwargs*)¶

class subprocess.**Popen**(*args*, *bufsize=-1*, *executable=None*, *stdin=None*, *stdout=None*, *stderr=None*, *preexec_fn=None*, *close_fds=True*, *shell=False*, *cwd=None*, *env=None*, *universal_newlines=None*, *startupinfo=None*, *creationflags=0*, *restore_signals=True*, *start_new_session=False*, *pass_fds=()*, ***, *group=None*, *extra_groups=None*, *user=None*, *umask=-1*, *encoding=None*, *errors=None*, *text=None*, *pipesize=- 1*)¶

These two functions take in very similar arguments, but there are two distinctions that are important to remember about when you should use each.

Run – This command is a blocking command. What this means is that your Python script runs the command and then waits for it to finish before moving to the next line of your Python code.

Popen – This command is non blocking. What this means is that your Python script runs the command and then comtinues on to your next line of your Python code without waiting. This is also called asynchronous code. You can later specify in your Python code if you want to wait for the completion of the command. This is also used if you want to pipe two commands together.

Let's see how this works, running the same command as before through the python interpreter.

```
>>> subprocess.run(['grep','Algeria','WorldDemographics.csv'])
202,Algeria,12,Country,912,0,1008303
203,Algeria,12,Country,912,1,1008303
…
301,Algeria,12,Country,912,99,1228
302,Algeria,12,Country,912,100,963
CompletedProcess(args=['grep', 'Algeria', 'WorldDemographics.csv'], returncode=0)
```

A few things you should notice. First, and this is confusing, to run a command, you supply a list of the name of the command followed by each argument of the command as a separate element of the list. Second, you will also see the results of the command printed to the screen along with a CompletedProcess line.

Part 2: Capturing output from a command using run

Very likely, you will want to capture the output from the command that you run to analyze further in the script. To do this, you will need to save the output to a variable AND use a separate argument for the run function called capture_output. Also, we will add a number to the search string so that we only return one line of the file to make it easier to explore what is returned

```
>>> output = subprocess.run(['grep','287,Algeria','WorldDemographics.csv'],
capture_output = True)
```

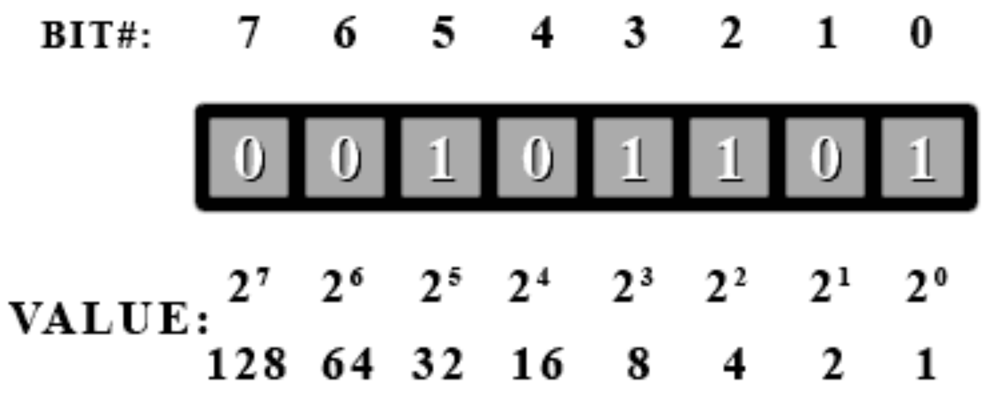
What is returned is an object that has parsed the command and output into a specialized object. Remember from previous lectures that the output of a program can go towards two different streams, the standard output or the standard error. This object also contains information on the command that was run as well as whether the command ran successfully.

```
>>> output
CompletedProcess(args=['grep', '287,Algeria', 'WorldDemographics.csv'], returncode=0,
stdout=b'287,Algeria,12,Country,912,85,48016\n', stderr=b'')

>>> type(output)
<class 'subprocess.CompletedProcess'>

>>> dir(output)
['__class__', '__class_getitem__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__weakref__', 'args', 'check_returncode', 'returncode', 'stderr', 'stdout']
```

However, note what is returned.

```
>>> output.stdout
b'287,Algeria,12,Country,912,85,48016\n'
>>> type(output.stdout)
<class 'bytes'>
```

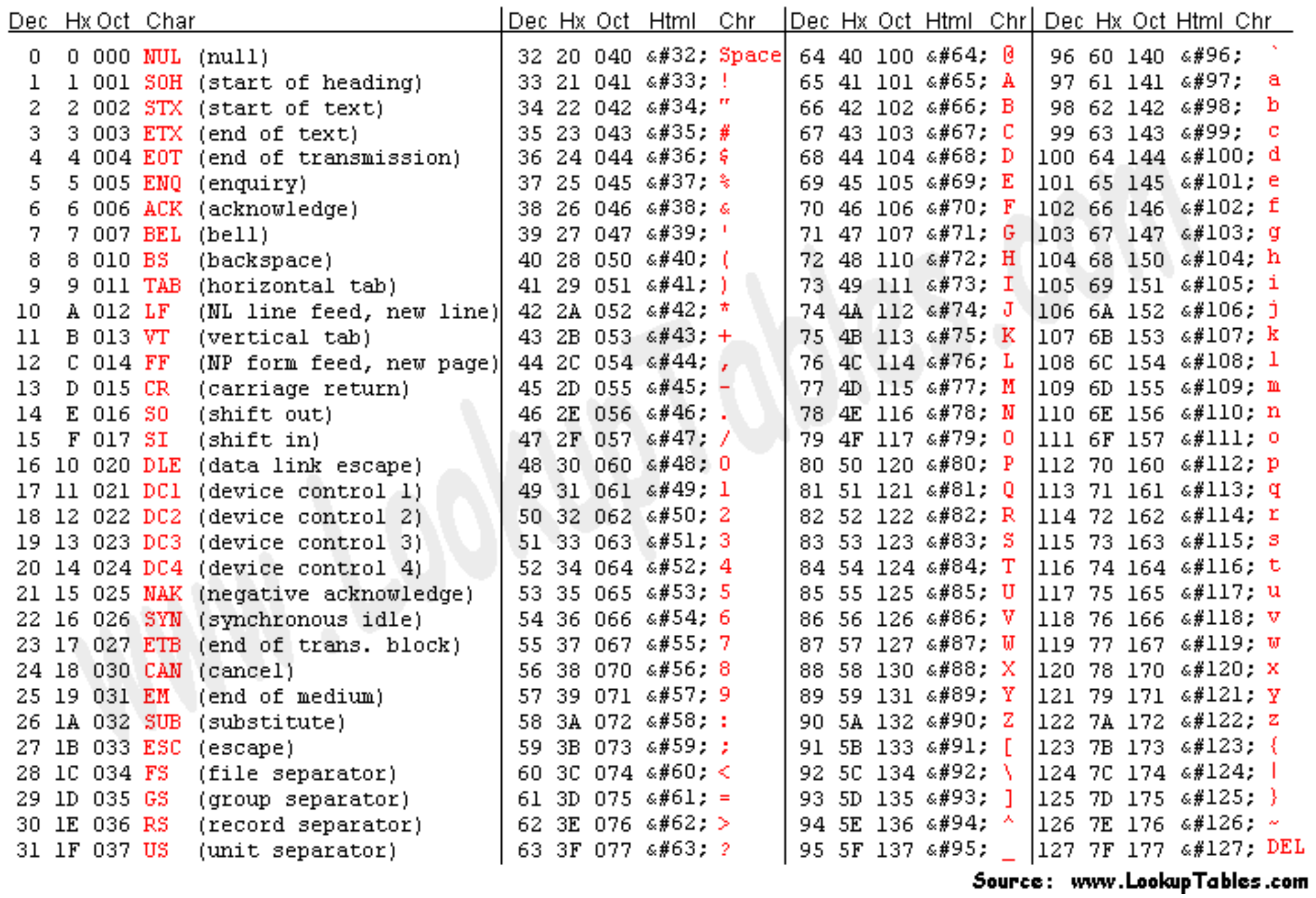
The output is a bytes object, which is essentially a list of bytes. A byte is a collection of 8 bits, which essentially means can take on the value of any number from 0 to 255.



We can see this by printing each element of the array.

```
>>> for element in output.stdout:
...    print(element)
...
50
56
55
44
65
108
103
101
114
105
```

How are these numbers related to letters? The answer is that there are defined tables that translate these number into different letters. For example, a common way to to translate a byte into a letter is using the ascii code:



Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	SOH (start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	STX (start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	ETX (end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	EOT (end of transmission)	36	24	044	&#36;	$	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	ENQ (enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	ACK (acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	BEL (bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	BS (backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	TAB (horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	LF (NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	VT (vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	FF (NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	CR (carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	SO (shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	SI (shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10	020	DLE (data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11	021	DC1 (device control 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12	022	DC2 (device control 2)	50	32	062	&#50;	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13	023	DC3 (device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14	024	DC4 (device control 4)	52	34	064	&#52;	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15	025	NAK (negative acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	SYN (synchronous idle)	54	36	066	&#54;	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	ETB (end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	CAN (cancel)	56	38	070	&#56;	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	EM (end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	SUB (substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	ESC (escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	FS (file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	|
29	1D	035	GS (group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	]	125	7D	175	&#125;	}
30	1E	036	RS (record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	US (unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

However, you might notice that ascii is quite English-centric. In is incapable of encoding letters common in other alphabets/languages for example. A more universal code is Unicode.

http://unicode-table.com/en/

You can access a subset of Unicode called ‘utf-8’ where the 8 stands for 8 bits. We can use both of these codes to translate the byte array into a string like so:

```
>>> output.stdout.decode('ascii')
'287,Algeria,12,Country,912,85,48016\n'
>>> output.stdout.decode('utf-8')
'287,Algeria,12,Country,912,85,48016\n'
```

You can also specify the decoding when you run the command using the optional encoding argument:

```
>>> output = subprocess.run(['grep','287,Algeria','WorldDemographics.csv'],
capture_output = True, encoding = 'utf-8')
>>> output
CompletedProcess(args=['grep', '287,Algeria', 'WorldDemographics.csv'],
returncode=0, stdout='287,Algeria,12,Country,912,85,48016\n', stderr='')
```

Part 3: Using Popen to pipe commands and run them asynchronously

Previously, we discussed using pipes to run two commands together. As an example, let's say we wanted to return the first three lines of the Algeria data. This is possible by piping the output of the grep command to the head command:

```
(base) ➜  ExampleData grep Algeria WorldDemographics.csv | head -3
202,Algeria,12,Country,912,0,1008303
203,Algeria,12,Country,912,1,1008303
204,Algeria,12,Country,912,2,1008303
```

There are a few things to consider when we run such a command.

1) The grep command runs and immediately outputs a line as soon as it finds a match. In other words, it does not wait to complete before sending its output to the standard out. For head to work, it must immediately be available to process the output, running at the same time as the grep command.
2) Head takes in its input from the standard input. The pipe redirects the standard output of the grep command to the standard input that head can read.

In order to run this in Python, we must use the Popen command, which allows us to run to commands at the same time like so:

```
>>> p1 = subprocess.Popen(['grep','Algeria','WorldDemographics.csv'], stdout =
subprocess.PIPE)
>>> p2 = subprocess.Popen(['head', '-3'], stdin = p1.stdout, stdout =
subprocess.PIPE)
>>> output = p2.communicate()
>>> output
(b'202,Algeria,12,Country,912,0,1008303\n203,Algeria,12,Country,912,1,1008303\n20
4,Algeria,12,Country,912,2,1008303\n', None)
```

Notice that we use PIPE twice. The first allows us to take the output of p1 and use it as input for p2. We also use it when we create p2. This allows us to capture the output of p2 and store it in the output variable.