GENERAL ASSEMBLY

DATA SCIENCE: COMMAND LINE

Dan Wilhelm, Lead Instructor
dan@danwilhelm.com

TERMINAL AND ENCODINGS

- Paths
- The System Path
- Bits and Bytes
- Text Encodings
- Unicode

EVER NEED HELP?

- Just type: `man <command name>`

Man uses `less`, a common Linux paginator.

```
q – quits
space – next page
/<search term> - searches
        (n – next match, N – last match)
g – go to start of file, G – go to end of file
```

PRESENT WORKING DIRECTORY

- At all times in the Terminal, you are inside of a directory.
- This is called the **present working directory**.

- **Linux/Mac**: `pwd` `Present Working Directory`

  `/Users/dan`

- **Windows**: The pwd is part of the command prompt, i.e.

  `C:\Users\dan\myfile.txt>`

PATHS: SPECIAL SYMBOLS

.	Current directory
..	Parent directory
/	File system root
~	Home directory

PATHS: SPECIAL SYMBOLS

.	Current directory
..	Parent directory
/	File system root
~	Home directory

Default home directories for user `dan`:

Mac: `/Users/dan/`

Windows: `C:\Users\dan\`

Linux: `/home/dan/`

PATHS: SPECIAL SYMBOLS

.	Current directory
..	Parent directory
/	File system root
~	Home directory

Default home directories for user `dan`:

Mac: `/Users/dan/`

Windows: `C:\Users\dan\`

Linux: `/home/dan/`

Useful Convention when coding

- Use a trailing slash when storing a directory or path.
- Do not use a trailing slash when storing a file.

PATHS: ABSOLUTE VS RELATIVE

Absolute paths start with `/`. They always start at the file system root, i.e.

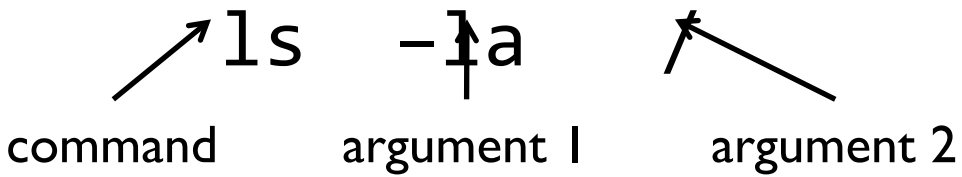
```
/Users/dan/
```

Relative paths start with anything else. They are relative to the present working directory, i.e.

```
./mydir/myfile.txt          ../otherdir/otherfile.txt
```

HOW THE TERMINAL WORKS

- When a user enters a command, the shell successively searches each directory in the system path until it finds an executable file with the same name.
- Then, it runs that executable with the given arguments.



HOW THE TERMINAL WORKS

- When a user enters a command, the shell successively searches each directory in the system path until it finds an executable file with the same name.
- Then, it runs that executable with the given arguments.

View the system path

```
>> echo $PATH

>> /Users/danwilhelm/anaconda/bin:/Library/Frameworks/Python.framework/Versions/3.4/bin:/Users/danwilhelm/.rbenv/shims:/Users/danwilhelm/.rbenv/shims:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/opt/X11/bin:/usr/local/git/bin:/Users/danwilhelm/.rvm/bin
```

In Windows, just type: PATH

Suppose you run `python` and the wrong version runs. Why did this happen? How can we debug it?

1) The command `which python` informs you which executable file is run when you run `python`.
2) If this is not what you expect, then your executable file may no longer exist. Or, a startup script may have prepended a directory to your PATH!
3) To resolve, prepend the correct directory to your PATH so that it is searched first. (Or, remove the insertion of the other directory.) To do this permanently, you would modify your startup script.

HOW THE TERMINAL WORKS

- When the Terminal is opened, several scripts are run. This includes `~/.bash_profile` (if using the Bash shell).

HOW THE TERMINAL WORKS

- When the Terminal is opened, several scripts are run. This includes `~/.bash_profile` (if using the Bash shell). For example:

```
export PS1="\[\033[36m\]\u\[\033[m\]@\[\033[32m\]\h:\[\033[33;1m\]\w\[\033[m\]\$ „
export CLICOLOR=1
export LSCOLORS=ExFxBxDxCxegedabagacad

alias ls="ls -G"

# added by Anaconda3 5.1.0 installer
export PATH="/anaconda3/bin:$PATH"
```

- This is how directories are added to PATH, colors are set up, etc.

Let's modify your PATH to add Anaconda. Before editing anything, make sure you modify them to reflect your Anaconda directory.

Windows: run Git Bash or Windows Subsystem for Linux.

1. First, find out where Anaconda is installed:

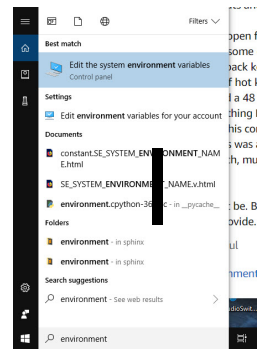
```
which conda
```

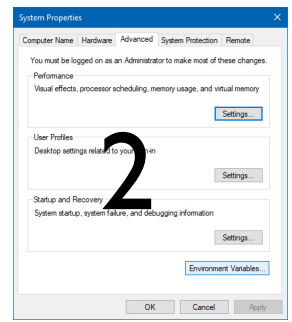
2. Based on where Anaconda is installed, append to `~/.bash_profile`:

```
export PATH="/anaconda3/bin:$PATH"
```

YOUR SYSTEM PATH (WINDOWS)

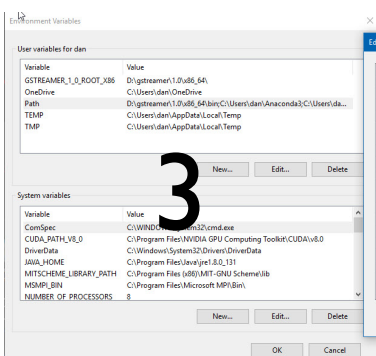
In Windows, you modify the system path via the GUI

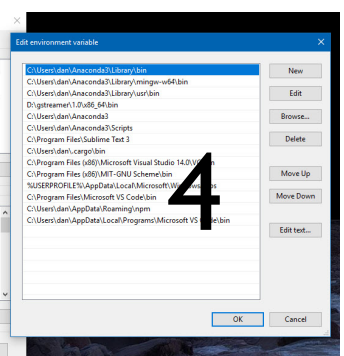




Windows:

1. Type `environment` in the lower-left search bar.
2. Select `Environment Variables...` near the bottom.
3. Click on the `Path` variable in the top box then select `Edit...`
4. Add the three directories shown and move them to the top.






```
C:\Users\<you>\Anaconda3\Library\bin
C:\Users\<you>\Anaconda3\mingw-w64\bin
C:\Users\<you>\Anaconda3\usr\bin
```

BITS AND BYTES

Bit

- Always in one of two states, typically referred to as 0 and 1.
- The atomic unit of memory.

Byte

- 8 bits
- The minimal addressable unit of memory.

BITS AND BYTES

- **1 bit:** can represent $2^1 = 2$ states (0 and 1).
- **2 bits:** can represent $2^2 = 4$ states (00, 01, 10, and 11).

We claim n bits can represent 2^n states.

- **n bits:** for each state in **(n-1) bits**, prepend a 0 or 1.
- Hence, there are twice as many states!
- And, $2*2^{n-1} = 2^n$.

```
00   -> 0(00) or 1(00)
01   -> 0(01) or 1(01)
10   -> 0(10) or 1(10)
11   -> 0(11) or 1(11)
```

YOUR MEMORY (AND DISK)

```
Memory:  01100001 01010011 10011111 01 …
Address: 0        1        2        3  …
```

YOUR MEMORY (AND DISK)

```
Hex:          0x61      0x53      0x9F
Decimal:       97        83       191
Memory:   01100001 01010011 10011111 01 …
Address:  0        1        2         3  …
```

YOUR MEMORY (AND DISK)

```
Hex:          0x61      0x53      0x9F
Decimal:       97        83       191
Memory:   01100001 01010011 10011111 01 …
Address:  0        1        2        3  …
```

YOUR MEMORY (AND DISK)

- All data is stored as 0s and 1s.
- So, we must come up with ways to encode different data types!

- **Unsigned integers** are encoded as binary (base 2).
- **Signed integers** are encoded using two's complement.
- **Floating point** is encoded using IEEE 754 (scientific notation).
- **Text** is encoded by mapping each possible state to a character.

GENERAL ASSEMBLY

DATA SCIENCE: INTRO TO PYTHON

Dan Wilhelm, Lead Instructor
dan@danwilhelm.com

INTRO TO PYTHON

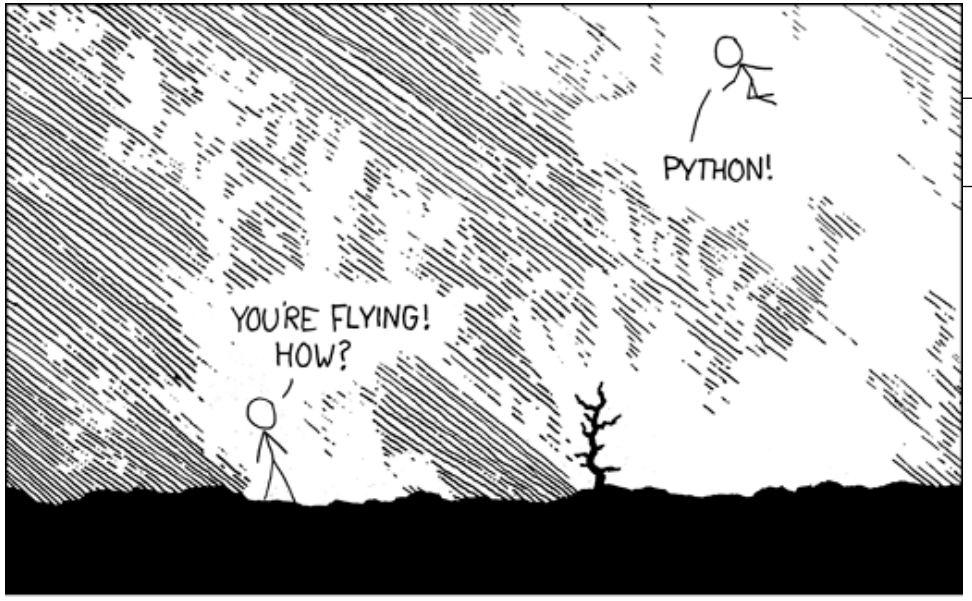
- What is Python?
- Python Fundamentals

PEP 20 (THE ZEN OF PYTHON)

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.
- Readability counts.

https://www.python.org/dev/peps/pep-0020/

Also see PEP 8 (Style Guide for Python Code)
https://www.python.org/dev/peps/pep-0008/



https://xkcd.com/353/

PYTHON LANGUAGE FEATURES

- Interpreted
- High-level
- Emphasizes readability

- Supports many programming paradigms – e.g. object-oriented, imperative, functional, and procedural.

- Dynamic typing
- Strongly typed
- Automatic memory management

WHY DO PEOPLE USE PYTHON?

- Open source (not locked in to one company)
- Clean syntax emphasizes readability
  - Great for scripting (interpreted & tolerant to errors)
- Modules and Packages
  - Large community – has many prewritten modules!
- Increased productivity
  - No compilation step, so edit-test-debug is fast
- Easy integration with C
  - Fast, pre-existing libraries can be used (called **bindings**)

WHEN SHOULD YOU NOT USE PYTHON?

- Performance matters.
- Low memory usage is important.
- Direct access to hardware required.
- Want OS-specific GUI features.
- Programming embedded systems.

PYTHON 2 VS PYTHON 3

> "Short version: Python 2.x is legacy, Python 3.x is the present and future of the language" - https://wiki.python.org/moin/Python2orPython3

Are you new to Python?

Learn version 3.

Are you writing a new program?

Research whether the libraries you want to use support Python 3. (They probably do.) If so, use Python 3!

Are you maintaining an old project?

Use the version of Python they use.

LEARNING PYTHON

- Official Tutorial
  https://docs.python.org/3/tutorial/index.html
- Google's Python Class
  https://developers.google.com/edu/python/
- Learn Python the Hard Way
  http://learnpythonthehardway.org/book/
- Most importantly, MAKE THINGS!

Insert class title

PYTHON FUNDAMENTALS

https://www.continuum.io/downloads

UPDATING CONDA PACKAGES

Update packages to latest stable Anaconda distro:

```
conda update anaconda
```

Update all packages to latest versions (warning: not recommended – will likely downgrade some to satisfy dependencies):

```
conda update --all
```

Upgrade to Python 3.7 from earlier version:

```
conda install python=3.7
conda install anaconda
conda clean --packages
```

VIRTUAL ENVIRONMENTS

Create an empty virtual environment w/ Anaconda installed:

```
conda env create –n <name> anaconda
```

Create a Python 2.7 virtual environment:

```
conda create -n py27 python=2.7 anaconda
```

List available virtual environments:

```
conda env list
```

CONDA IS A PACKAGE MANAGER

- **conda** – A package management system used to install and manage software packages written in Python. It has benefits over `pip`:
  - Dependency checking!
  - Downloads binaries instead of compiling from source (although it can)
- Always try to install packages using `conda` first before `pip`.

conda search <part of package name>

conda install <package name>

conda info <package>

(Frees GBs of space!)

conda clean --all

PIP INSTALLS PACKAGES

- **pip** – A package management system used to install and manage software packages written in Python.
- If a package is available on Anaconda, install via conda.
- If a package is not available on Anaconda:

```
pip install money
pip uninstall money
```

DATA TYPES

- Booleans (True or False)
- Numbers
- Strings
- Lists
- None
- Tuples
- Sets
- Dictionaries

ADDED BY PANDAS/NUMPY:

- ndarray (from numpy)
- Series (from pandas)
- DataFrame (from pandas)

PYTHON IDENTIFIERS

- Python is not as expansive as you may think.
- Every identifier you see (without any imports) is either an operator, reserved word, or built-in! So, there are a limited number of things to know.

PYTHON IDENTIFIERS

- Python is not as expansive as you may think.
- Every identifier you see (without any imports) is either an operator, reserved word, or built-in! So, there are a limited number of things to know.

- An identifier is:

A single letter or underscore, followed by zero or more letters, numbers, or underscores.

```
(letter | _) (letter | number | _)*
```

RESERVED WORDS/KEYWORDS

- Reserved words require special syntax to be used around the word – their syntax is unique compared to the rest of Python.
- Reserved words cannot be used as variables.

```
False      class      finally    is         return
None       continue   for        lambda     try
True       def        from       nonlocal   while
and        del        global     not        with
as         elif       if         or         yield
assert     else       import     pass
break      except     in         raise
```

https://docs.python.org/3/reference/lexical_analysis.html#keywords

BUILT-IN FUNCTIONS

		Built-in Functions		
abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

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

HOW TO CODE

When coding, you only have a limited number of options:

- Use a built-in function.
- Use a method of the object.
- Import a module.
- Write your own function.

STATEMENTS VS EXPRESSIONS

An **expression** is valid code that evaluates to a value:

```
>>> 10 + 7
>>> range(10)
```

A **statement** is a line of code that performs an action, e.g.:

```
>>> print(10 + 7)
>>> x = 10 + 7
```

Programs are collections of statements.

CONDITIONALS

```
>>> donuts = 5
>>> if donuts < 2:
...     print(“Good job!”)
... else:
...     print(“You ate too many donuts.”)
```

- According to PEP 8, use 4 spaces per indentation level.
- Use ‘=‘ for assignment and ‘==‘ for comparison.
- Name your variables well! Why is “donuts” a poor name?

STRINGS

```
>>> x = input(“Exit program? (yes/no)”)
>>> if x.lower() == “yes”:
...     print(“Exiting the program ...”)
```

- Notice the method call. What does it do? Why is it done?
- Use ‘=‘ for assignment and ‘==‘ for comparison.
- Is x a good variable name? What should it be?
- ‘ vs “ vs “””
- Indexing characters.

LISTS VS TUPLES

- A list contains ordered data, typically of the same data type:

```
>>> x = [“Tim”, “Sandy”, “Martin”, “Shawna”]
>>> print(x[0])
```

- A tuple contains ordered groups of variables, often of different data types:

```
>>> x = (“Tim”, 5)   # Note (name, age) describe one person
>>> print(x)
```

LOOPING

- Given a list or tuple, we often want to do something with each member. To do this, we loop through the list:

```
>>> names = [“Tim”, “Sandy”, “Martin”, “Shawna”]
>>> for name in names:
...     print name
```

- The indented code in the for block is run for each item in the list. Each time the indented code is run, *name* refers to the next item in the list. Note that *name* can be any name.

FUNCTIONS

- A function allows us to take complex code and refer to it in an easy way. For example, "x % 2 == 1" is hard to understand by beginners if encountered in a program. Hence, to make the program easier to read, we refer to what the code does in English, as part of a function call that returns a value:

```
>>> def is_odd(x):
...     return (x % 2 == 1)
```

- What is a better name for x?

HOW TO CODE

When coding, you only have a limited number of options:

- Use a built-in function.
- Use a method or operator of the object.
- Import a package.
- Write your own function.

INTERACTING WITH THE COMMAND LINE

- Retrieving command-line arguments is easy!

```
import sys
sys.argv    # ARGument Vector (list)

Example: python test.py 1 2 3
sys.argv: ['test.py', '1', '2', '3']
```

INTERACTING WITH THE COMMAND LINE

- Reading from stdin is just like reading a file!

```
import sys
for line in sys.stdin:
    print(line.strip())
```

FILES & UNICODE

FILES (OLD WAY)

```
file_in = open('test.txt', 'r', encoding='utf-8')

for line in file_in:
    print(line)

file_in.close()
```

FILES (OLD WAY)

```
file_in = open('test.txt', 'r', encoding='utf-8')


for line in file_in:
    print(line)


file_in.close()
```

common file modes

'r' – read
'w' – write
'a' – append
'b' – binary

FILES (NEW WAY) – READING

```
# ‘with’ insures the file is closed if an exception occurs
with open(‘test.txt’, ‘r’ , encoding=‘utf-8’) as fin:
    for line in fin:
        print(line)

# Same thing, except the ‘with’ block ensures the file is
# closed – even if an exception occurs!
```

FILES (NEW WAY) – WRITING

```
# Write list of strings ‘lines’ to the file
with open(‘test.txt’, ‘w’) as fout:
    for line in lines:
       fout.write(line + “\n”)
```

ENCODINGS

- All files are just sequences of bytes (aka 8-bit numbers). So, all files look alike.
- Hence, the operating system needs a way to know how to interpret a file's contents.

ENCODINGS

- All files are just sequences of bytes (aka 8-bit numbers). So, all files look alike.
- Hence, the operating system needs a way to know how to interpret a file's contents.
- There are only two main ways to do this:
  1. The file extension provides a hint at how to interpret the file, e.g.:

     .jpeg .htaccess .txt .docx
  2. The beginning of the file could provide clues, e.g.:

     all JPEG files start with the sequence: 255, 216

ASCII VS UNICODE

- Text files (.txt, .py, .html) have historically used the 7-bit **ASCII** encoding
- However, ASCII only supports the English alphabet. So, many other text encodings were created.
- Collectively, these alternative encodings are known as **Unicode**.

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

Source: www.LookupTables.com

ASCII

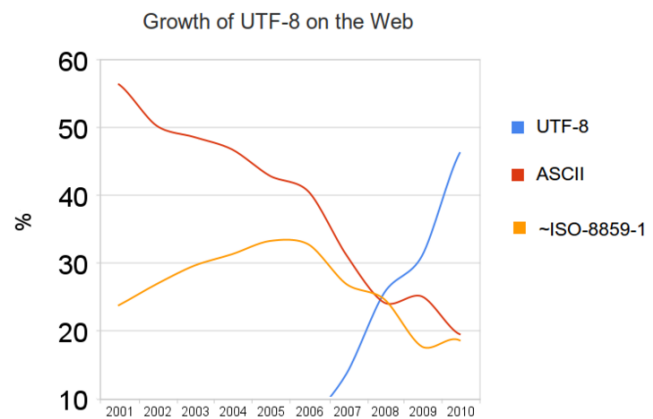
A 7-bit text encoding

ASCII VS UNICODE

- *Problem*: Nowadays, text files could be encoded via ASCII or one of many Unicode encodings.
- However, there is no way to infer how to interpret the numbers -- the filename extensions are all the same, and text files immediately begin with the text content!

ASCII VS UNICODE

- For example, the most popular encoding is UTF-8. It relies on ASCII being 7 bits and uses the 8th bit as a flag to indicate a special character.
- Hence, it is backwards compatible with ASCII. An ASCII file is also a UTF-8 file! (Assuming the 8th bits are all zero.)



ASCII VS UNICODE

- For example, the most popular encoding is UTF-8. It relies on ASCII being 7 bits and uses the 8th bit as a flag to indicate a special character.
- Hence, it is backwards compatible with ASCII. An ASCII file is also a UTF-8 file! (Assuming the 8th bits are all zero.)
- If an ASCII file has a non-zero 8th bit, this will throw a Python exception saying it is invalid ASCII!
- If a UTF-8 encoded-file does not follow the UTF-8 rules, then it will also throw a Python exception, e.g. if Byte 1 is 110xxxxx, the next byte *must be* 10xxxxxx, where ‘x’ is any bit.

COMMON UNICODE ENCODINGS FOR OPEN()

‘ascii’	ASCII
‘latin-1’ (‘iso-8859-1’)	ASCII w/ extended ASCII
‘utf-8’	variable-length encoding (min. 1 byte)*
‘utf-16’	var-length encoding (min. 2 bytes)

* backward-compatible with ASCII

Also:

- \u<CODE POINT> is the escape char for 2-byte Unicode chars
- \U<CODE POINT> for 4-byte Unicode chars

UTF-8: A VARIABLE-LENGTH ENCODING

How to store different Unicode code words

	1st byte	2nd byte	3rd byte
U+0000 – U+007F	**0nnn nnnn**		
U+007F – U+07FF	110n nnnn	10nn nnnn	
U+0800 – U+FFFF	1110 nnnn	10nn nnnn	10nn nnnn

Note the backward-compatibility with ASCII! (bolded)

ASCII VS UNICODE

- If we ask Python to interpret a file using a certain encoding and the file does not follow these rules, then an exception will be thrown.
- This is unfortunately the only way to know whether we guess the encoding correctly – attempt to interpret the file numbers in one way and see if it follows the encoding's rules.

ASCII VS UNICODE

- If we ask Python to interpret a file using a certain encoding and the file does not follow these rules, then an exception will be thrown.
- This is unfortunately the only way to know whether we guess the encoding correctly – attempt to interpret the file numbers in one way and see if it follows the encoding's rules.
- So, it is preferred to try opening the file using different encoding. If this does not work, you can choose to ignore errors.
- Note: When you open a file in an IDE or text editor and save it, the IDE may resave the file in a different encoding!

PYTHON + DATA SCIENCE TRICKS

LISTS VS TUPLES

- Lists: Mutable (can be altered), typically homogenous values
- Tuples: Immutable, typically groups of items that go together

```
>>> people = [('Tim', 32), ('Sandy', 45)]
>>> points = [(0,0,0), (5,4,1), (7,7,8)]
>>> menu_item, price = ('Burger', 2.99)   # unpacking
>>> def max_population(cities):   # multiple return values
        ...
        return (city_name, population)
```

QUOTES

- ‘vs “ ← same – both support escape characters (“\n“)
- ‘‘‘ ← triple quotes – allows actual newlines

- \ ← if at the end of a non-quoted line of code, allows you to split the line of code (there is an invisible newline after it)

```
>>> names = “Mike Wallace\nClara Simmons”
>>> names.split(“\n”)
>>> names.replace(“\n”, “, “)
```

STRING FORMATTING

```
“Person #{} is {}”.format(2, ‘George’)

“Person #{num} is {name}”.format(num=5, name=‘Henry’)

“Your price will be ${price:.2f}.”.format(price=4.5127)
```

MODULES

```
import math
>>> math.sqrt(5)

from math import sqrt
>>> sqrt(5)

from math import *
>>> sqrt(5)
>>> tan(5)
```

LIST COMPREHENSIONS

```
cubes = []
for num in range(100):
    cubes.append(num**3)
```

BECOMES

```
cubes = [num**3 for num in range(100)]
```

LIST COMPREHENSIONS

```
cubes = []
for num in range(100):
    if num % 2 == 0:
        cubes.append(num**3)
```

BECOMES

```
cubes = [num**3 for num in range(100) if num % 2 == 0]
```

NOTE you can use these for filtering!

ENUMERATE – WHEN YOU NEED A LOOP INDEX

```
index = 0
for person in people:
        print(“Person #{} is {}”.format(index, person))
        index += 1
```

BECOMES

```
for index,person in enumerate(people):
        print(“Person #{} is {}”.format(index, person))
```

DATES AND TIMES

- `datetime.date -> year, month, day`
- `datetime.time -> hour, minute, second, microsecond, tzinfo (TimeZone INFO)`
- `datetime.datetime -> year, month, day, hour, minute, second, microsecond, tzinfo`

- `datetime.timedelta – difference between dates/times`

EXCEPTIONS

```
try:
        num = int('not an int')
except:   # catches ALL exceptions
        print('Exception caught!')


try:
        num = int('not an int')
except ValueError:   # catches the ValueError exception
        print('Exception caught!')
```

LISTS VS TUPLES

- Lists: Mutable (can be altered), typically homogenous values
- Tuples: Immutable, typically groups of items that go together

```
>>> people = [('Tim', 32), ('Sandy', 45)]
>>> points = [(0,0,0), (5,4,1), (7,7,8)]
```

LISTS VS TUPLES

- Lists: Mutable (can be altered), typically homogenous values
- Tuples: Immutable, typically groups of items that go together

```
>>> people = [('Tim', 32), ('Sandy', 45)]
>>> points = [(0,0,0), (5,4,1), (7,7,8)]
>>> menu_item, price = ('Burger', 2.99)   # unpacking
```

LISTS VS TUPLES

- Lists: Mutable (can be altered), typically homogenous values
- Tuples: Immutable, typically groups of items that go together

```
>>> people = [('Tim', 32), ('Sandy', 45)]
>>> points = [(0,0,0), (5,4,1), (7,7,8)]
>>> menu_item, price = ('Burger', 2.99)   # unpacking
>>> def max_population(cities):   # multiple return values
        ...
        return (city_name, population)
```

FILES

```
# ‘with’ ensures the file is closed if an exception occurs
with open(‘test.txt’, ‘r’) as fin:
    for line in fin:
        print(line)

# Write list of strings ‘lines’ to the file
with open(‘test.txt’, ‘w’) as fout:
    for line in lines:
       fout.write(line + “\n”)
```

OTHER STUFF

- Sets
- Using dictionaries for uniqueness/histograms

MODULES FOR SCRAPING & APIS

```
pip install requests
```

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

```
pip install beautifulsoup4
```

http://www.crummy.com/software/BeautifulSoup/bs4/doc/