Week 9:
Encoding Text
and a quick glance at NymPy

你好?

سالام

हिंदिसटन

A À Å B C D E É G H I Î J K L M N Ñ Ø P Q R S ẞ T Ü V W X Y Z

Tonight's agenda

The Second Half!
Congratulations on completing part 1.

Working with text, binary data. NumPy.
More NuyPy.
Data Analysis with Pandas
More Data Analysis with Pandas
Group Work
... and your final quiz
Code Testing and ... your final project showcase!

Schedule …

Class 9 - Working with text and binary

- Encoding
- Unicode Strings
- Formatting
- RegEx
- Binary
- File I/O
- Structured Files

10: NumPy

11: Data Analysis with Pandas

12: Plotting & Visualization

13: Pandas aggregation & group operations

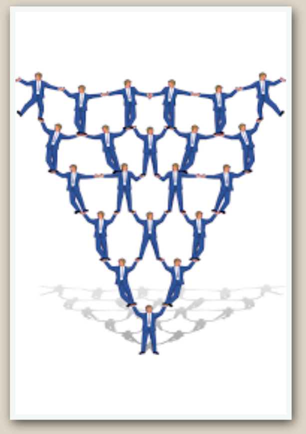
14: Testing & Wrapup.

Any questions about the showcase?

- *Some questions that you may address …*
  - Explain your project at a high level
  - Share your screen, run your code, a couple of slides if you want, … show off what you've done!
  - Open the code and share to discuss …
    - what classes did you use to solve your problem?
    - what were the major challenges of your implementation?
  - *Please* practice a *brief* (*about 5 min, tops*) *presentation that* communicates *your project to others. Very important skill.*

Moving on … Up and down levels of abstraction

- *We've traversed the levels of abstraction … up and down …*
  - Fundamental types: ints, floats … [aka primitives]
  - Container objects: lists, strings
  - Classes



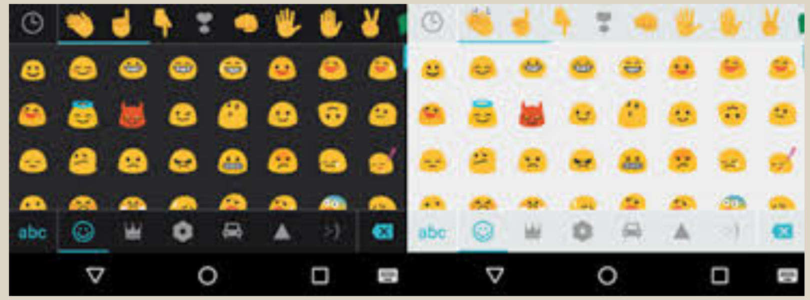
- Now … drill down to characters and bytes [oh, boy!]

Converting the text "hope" into binary

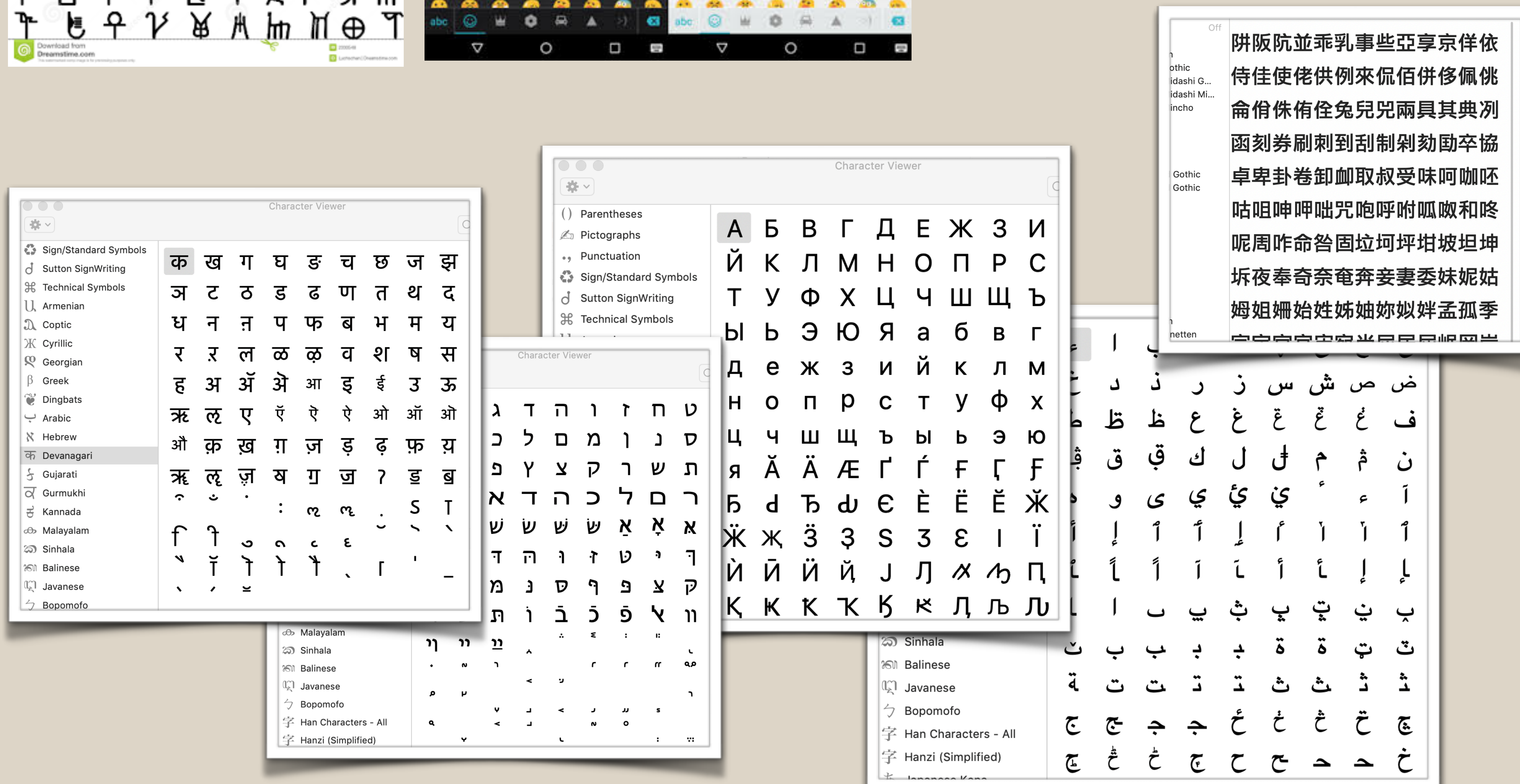
Characters:	h	o	p	e
ASCII Values:	104	111	112	101
Binary Values:	01101000	01101111	01110000	01100101
Bits:	8	8	8	8

ComputerHope.com

The challenge? Translate icons/type to code & back



http://unicode.org and check out the codesheets.



Translating encoding schemes …

- *Everything is stored in binary* (0, 1)
- *At some point, data are translated into this common format.*
- *BTW, note that how the data are stored on the hard drive is the "internal reflection" of the data (usually as UTF-8); and then there's the "external reflection" of the data (what's shown on the screen, often UTF-8, but could be win-1285, MacRoman, koi-8, etc.!).*
  - https://en.wikipedia.org/wiki/Character_encoding
- *Keep in mind: it's all just data! We can call "X" the same thing in a variety of "dialects" … as you'll see! (grin)… But first things first …*

Encoding Schemes … before we press on

- *Binary (base 2;* 0 or 1)
- *Octal (base 8*)
- *Decimal (base 10; back to* 0,1,2,3,4,5,6,7,8,9)
- *Hexadecimal (base 16;* 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F)
- This leads to various expressions: e.g., "byte" [8-bits], "nibble" [4-bits]; "multibyte" [unicode, ISO-639-x, utf-x [8, 16, 32, 64, even 132.]].
  - Check out: http://unicode.org/charts/charindex.html for the names!
- BTW, when data don't appear on screen correctly it is often an encoding mis-match error between the data stream and the output device's encoding settings. In word processing programs and in CJK and other language groups, byte-shifting is critical to storage and retrieval.

Encoding Schemes

- *Most end-users think about what they type ... but most professional/industrial standards* require *UTF-8. (https://www.w3schools.com/charsets/ref_html_utf8.asp). There are hundreds of encoding schemes - from Big5, TwinBridge to IIS, JIS, MacRoman, win-1285, utf-8, utf-16, etc.!*

- *Base 10 example:*

  - 0000 = 0 [zero in all places, 0, 0, 0, 0]
  - 0001 = 1 [1 x 2^0] 1 in the one's place *nb: take the bite * 2...
  - 0010 = 2 [1 x 2^1] 1 in the two's place
  - 0100 = 4 [1 x 2^2] 1 in the four's place
  - 1000 = 8 [1 x 2^3] 1 in the eight's place
  - 1100 = 12 [1 x 2^4] 8 + 4 ... on in the eight's place and one in the four's ... and so on...

In binary every number takes one of 2 values 0 or 1)

The places are multiples of 16 -> 1, 2, 4, 6, 8, 10

Encoding Schemes, con't.

- The choice of base 16 comes directly from the use of phrases binary strings; first 3 (Octal) and then 4 digit (Nybble, aka: hexadecimal)

3 digit binary phrase (Octal)

000=0, 001=1, 010=2, 011=3, 100=4, 101=5, 011=6, 111=7

These 8 numbers represent one Octal digit that can take values 0-7

Similarly the 4 digit binary phrase ("nibble") has 16 values and represents one hexadecimal digit (0-F)

Encoding Schemes, con't.

Computers also use hexadecimal which is base 16,

– 000 is 0   zero in all places

– 005 is 5   (5 x 2**0) one in the 1s place

– 050 is 16   (5 x 16**1) one in the 16s place

– 500 is 1280 (5 x 16**2) one in the 256s place

	Location					
	6	5	4	3	2	1
Value	1048576 (16^5)	65536 (16^4)	4096 (16^3)	256 (16^2)	16(16^1)	1 (16^0)

In hexadecimal every number takes one of 16 values coded as 0-F
0 1 2 3 4 5 6 7 8 9 A B C D E F

ASCII?!

ASCII uses 7 binary bits

$2^7 = 128$ characters

Please note that there's also 8-bit ascii.

[https://www.sciencebuddies.org/science-fair-projects/references/table-of-8-bit-ascii-character-codes]

There are other encoding systems

What are some?

* Often incompatible *

This is a partial table of characters in 8-bit ASCII.

Decimal	Octal	Hex	Binary	Value	
000	000	000	0000 0000	NUL	"null" character
001	001	001	0000 0001	SOH	start of header
002	002	002	0000 0010	STX	start of text
003	003	003	0000 0011	ETX	end of text
004	004	004	0000 0100	EOT	end of transmission
005	005	005	0000 0101	ENQ	enquiry
006	006	006	0000 0110	ACK	acknowledgment
007	007	007	0000 0111	BEL	bell
008	010	008	0000 1000	BS	backspace
009	011	009	0000 1001	HT	horizontal tab
010	012	00A	0000 1010	LF	line feed

Unicode, 1

Unicode encodes 120k characters

Modern and ancient languages, math

UTF-8

Compatible with Unicode

Python 3 has native support for unicode

Note: Windows 10 may create files using UTF-10. For example, if you try to create a file using the command line: echo "test" >> test.txt

If you run into encoding issues, this may be the cause.

Unicode, 2

Python 3 has native support for unicode - All strings are Unicode strings!

>>> "\u0047\u0072\u0072\u0021" == 'Grr!' True

>>> "\u0047\u0072\u0072\u0021" == 'GRR!' False

[Optional note: Some characters, particularly historical ones and ligatures, may be software or O/S dependent. Accessing these characters requires knowing and being able to read the "GID": graph identification number.]

Unicode, 3

Every unicode value has a standard name

Unicodedata.name() to get name from a value

Value can be literal "B" or unicode value "/u0042"

Returns "LATIN CAPITAL LETTER B"

Benefit? *Every* character can be identified *uniquely!*

You can often paste exotic characters

We can encode a text string in unicode using encode('utf-8'). Other options are available.

To decode unicode, use decode('utf-8')

b denotes **bitwise encoding**
\x means "hexadecimal"

```
>>> s.encode('utf-8')
b'\xe3\x88\xb2'
>>> s.encode('unicode_escape')
b'\\u3232'
```

Encoding & Decoding: (codec)

Not all characters can be represented in each encoding scheme! You can specify how to handle this

1) Replace with blank ('?')
2) XML friendly
    1) https://msdn.microsoft.com/en-us/library/aa468560.aspx
3) Unicode Escape (backslash)

Encoding & Decoding | Methods & Packages

Try the following commands:

- Unicodedata package
- `encode (), decode()`
- `type (), len()`

Regular Expressions (RegEx) | Finding Patterns

1. re.compile()   # compile a search string
2. re.search()   # gets the first match
3. re.match()   # extract match - if at beginning
4. re.split()   # split on matches
5. re.sub()   # substitute on matches
6. re.findall()   # get all matches as list
7. .group()   # used after matching to pull out groups

RegEx | Special characters & specifiers

1. . # any character 1 place
2. * # any number of char
3. ? # any character optional
4. [0-9] , /d # any digit
5. [a-z] # any letter lowercase letter
6. /w # any alpha-numeric char
7. r’ ’ # the raw string literal

Pattern	Matches
\d	a single digit
\D	a single non-digit
\w	an alphanumeric character
\W	a non-alphanumeric character
\s	a whitespace character
\S	a non-whitespace character
\b	a word boundary (between a \w and a \W, in either order)
\B	a non-word boundary

RegEx | Specifiers

<table>
<tr><th>Pattern</th><th>Matches</th></tr>
<tr><td>abc</td><td>literal abc</td></tr>
<tr><td>(expr)</td><td>expr</td></tr>
<tr><td>expr1 | expr2</td><td>expr1 or expr2</td></tr>
<tr><td>.</td><td>any character except \n</td></tr>
<tr><td>^</td><td>start of source string</td></tr>
<tr><td>$</td><td>end of source string</td></tr>
<tr><td>prev ?</td><td>zero or one prev</td></tr>
<tr><td>prev *</td><td>zero or more prev, as many as possible</td></tr>
<tr><td>prev *?</td><td>zero or more prev, as few as possible</td></tr>
<tr><td>prev +</td><td>one or more prev, as many as possible</td></tr>
<tr><td>prev +?</td><td>one or more prev, as few as possible</td></tr>
<tr><td>prev { m }</td><td>m consecutive prev</td></tr>
<tr><td>prev { m, n }</td><td>m to n consecutive prev, as many as possible</td></tr>
<tr><td>prev { m, n }?</td><td>m to n consecutive prev, as few as possible</td></tr>
<tr><td>[abc]</td><td>a or b or c (same as a|b|c)</td></tr>
<tr><td>[^ abc]</td><td>not (a or b or c)</td></tr>
<tr><td>prev (?= next)</td><td>prev if followed by next</td></tr>
<tr><td>prev (?! next)</td><td>prev if not followed by next</td></tr>
<tr><td>(?<= prev) next</td><td>next if preceded by prev</td></tr>
<tr><td>(?<! prev) next</td><td>next if not preceded by prev</td></tr>
</table>

RegEx | Basic Examples

```
middle_pattern = re.compile("that is")
m = middle_pattern.search("that is")

if m:
    print(m.group())
```

```
that is
```

```
n_pattern = re.compile("n") #Lets find all of the n's
m = n_pattern.findall(source)
print("Found", len(m), "matches")
print(m)
```

```
Found 2 matches
['n', 'n']
```

RegEx | Phone Number Example

Compact version

```
phone_number_pattern = re.compile(r'\d{3}-\d{3}-\d{4}')
```

expanded version

```
e(r'[0123456789]{3}-[0123456789]{3}-[0123456789]{4}')
```

RegEx | Matching Groups

```
phone_number_pattern = re.compile(r'(\d{3})-(\d{3}-\d{4})')
m = phone_number_pattern.search(large_source)

if m:
    print(m.group())
    print(m.groups())
```

```
650-555-3948
('650', '555-3948')
```

```
phone_number_pattern = re.compile(r'(?P<areacode>\d{3})-(?P<number>\d{3}-\d{4})')
m = phone_number_pattern.search(large_source)

if m:
    print(m.group("areacode"))
    print(m.group("number"))
```

```
650
555-3948
```

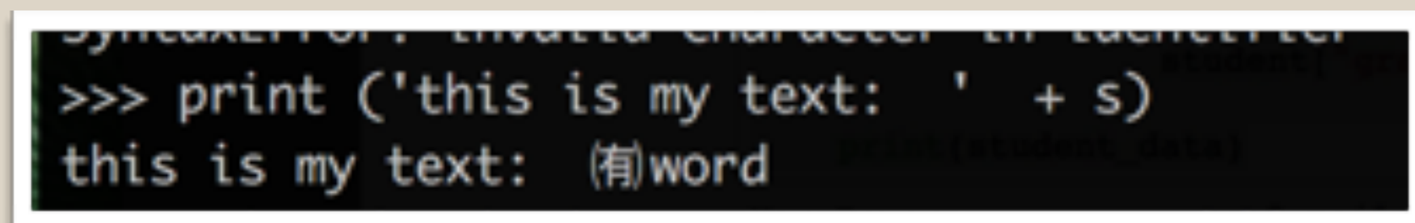
Text Output | Basics

Consider:

>>> s=′㈲word′

Simple concatenation

print (‘this is my text: ’ + s)



Text Output | The Basics | Oldstyle

The old s(%) style

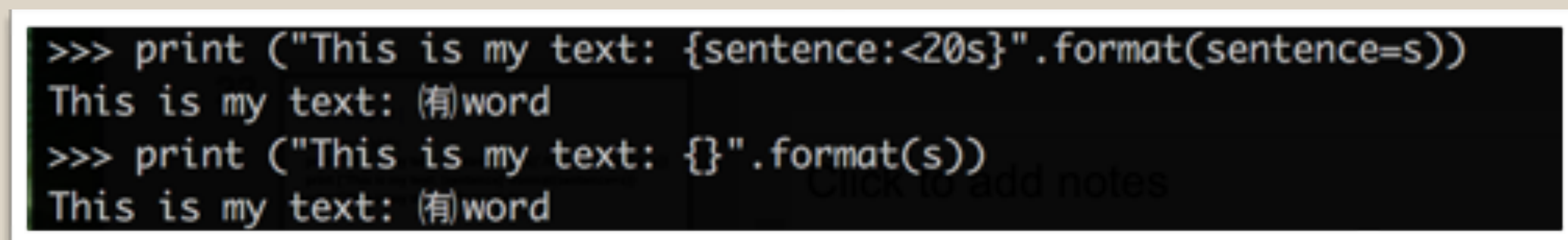
>>> print (‘this is my text: %10s ’ % (s))

```
>>> print ('this is my text: %s '  %  (s))
this is my text: (有)word
>>> print ('this is my text: %10s '  %  (s))
this is my text:       (有)word
```

Text Output | The Basics | New Style

The new {} style

```
print ("This is my text:
{sentence:<20s}".format(sentence=s))
print ("This is my text: {sentence}".format(sentence=s))
print ("This is my text: {}".format(s))
```



Basic Patterns | Python Functions

1. .open(file, mode),
   a. **open modes (‘wt’, ‘rt’, ‘at’, ’rb’, ‘wb’)**
2. **Action on file:**
   a. write(), read(), readlines(), readline()
3. .close()

Loading **Files** | Python Functions

1. .open(file, mode), .close()
2. open modes (‘wt’, ‘rd’, ‘at’, ‘rb’, ’wb’)
3. with() # you don’t need to close this one
4. read()
5. readlines() # reads all lines as a list
6. readline() #reads one line in at a time

NumPy

Quick **NumPy**

http://www.numpy.org

NumPy gives you the ability to work with **n-dimensional** arrays of numeric data of many types.

Pandas is built on top of NumPy and provides a more user friendly experience. There, we work with a “dataset” and include non-numeric variables.

Understanding NumPy is critical to understanding more advanced packages.

A basic understanding of NumPy will deepen your understanding of Pandas.

NumPy offers vectorized operations

** But see this: https://timothyhelton.github.io/pandas_best_practices.html

NumPy | Python Functions

http://www.numpy.org

1. np.array()
2. np.arange(), np.linspace()
3. np.min(), np.max(), np.std(), np.var()
4. np.argmax(), np.argmin()
5. np.shape(), np.reshape()
6. np.zeros()
7. np.random.seed(), np.random.random_integers()
8. np.vstack(), np.hstack()
9. Dealing with n-dimensions: “axis = “ (o or 1)