Week 4: Strings, Characters & Files

Honestly it will be fun but so diverse there can’t be a description

What do we need file I/O

In most programs we require some sort of non-volatile save. This means that our data need to be stored in a file which can be recovered, read by the code and, if need be, overwritten.

Case 1: Runtime Data Saves

In the case of many programs, it is useful to retrieve saved data from files. In the case of Video Games or human interface programs, say a contact book, we retrive data at the start of the program.

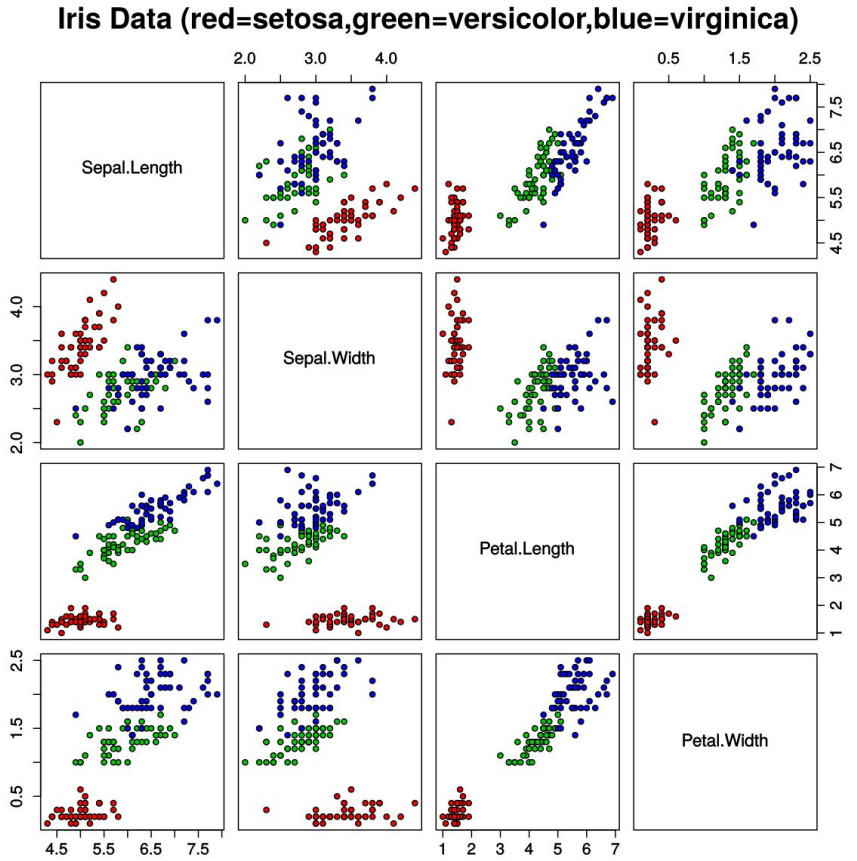
This can also be the case for low interface programs (something that runs without any user input) where we might have a configuration file to input some environment variables.

A character location can be represented as a variable

Case 2: Large Data Entries

Notice that so far all our data input for our programs have been in the form of manual typing. What if we wanted to work with a data set that had say 1,000,000 data points?

Typing would be unreasonable. Therefore, we can use File Input for reading large datasets that our program will be using.



Text Files

The Text file format (.txt) is what we will be using today for file I/O.

Text files are encoded using the UTF-8 (Unicode) format which supports 1,112,064 characters and is backwards compatible with ASCII.

Notice that when writing a line in a text file, it doesn't automatically move onto the next line? "New line" is a special ASCII character represented by the "escape sequence" '\n'.

Opening Files

We use the built in *open()* function to read a file. This function takes two arguments, the file name (string) and the mode (string) which can be:

- “r” read, opens a file. Returns Error if the file doesn’t exist
- “a” append, opens a file without emptying it
- “w” write, opens a file and clears it. Creates it if it doesn’t exist
- “x” create.

open(file_name, mode)

Mode can be omitted and “r” is the default value.

Reading Files

To actually read a file we use the *read()* function which returns the entire text file as a string object. The open function returns a file object so we need to use read after. The function takes a single argument which is *size*. For text input (.txt, .csv etc.) this is a value that represents number of characters up until to read.

read(size)

The default value is unlimited characters. However this can cause issues when dealing with systems that have small memory.

Strings

In Comp. Sci, we can treat Strings just like Numbers and access them like lists. In Python, Strings are treated as an array of characters which is very useful for us.

Just like in other languages, Strings are immutable which means they cannot be changed in any way. We can however re-append a different string to the same name.

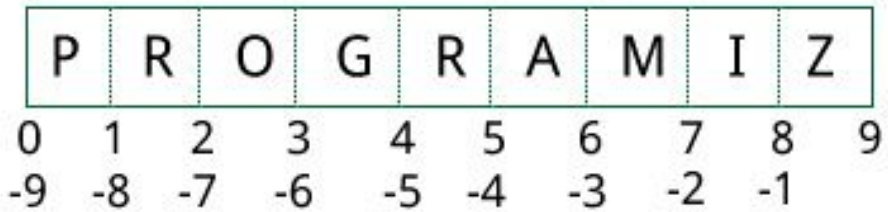
We can also delete strings from memory using the keyword *del*

String Indexing

We can also treat strings like arrays of characters in order to see/use certain parts of the string. This can be done as follows:

Character Indexing: string[index]
Range indexing: string[from : to]

Don’t forget that indexing starts at 0!



String Iteration

If you remember from Loops, we can iterate through a structure using the following expression:

For i in structure

We can do the same thing for strings with each i being a string of length 1 i.e. a letter.

For letter in string_name

However, it is recommended that when trying to work with individual letters, a string to list conversion is preferable.

Small Stuff

We can concatenate strings by using the ‘+’ modifier. This is done as such:

string1 + string2

We can also check if a string is a subset of another string using the following:

string1 in string2

Which of course is a **boolean** expression.

Name	Input	Output	Comments
len()	String	Int	Starts at 1
lower()	String	String	Turns to all lowercase
upper()	String	String	All to uppercase
find()	String	Int	Returns the number a string is subset
join()	String	String	Adds the str specified after each item in the main str
replace()	String (x2)	String	Replaces sub.s with specified string
split()	String	String	Splits a string at some operators

This is only a few of the functions

The most important. Format()

Format() allows us to have strings that have fields which can change dynamically.
By specifying the field using { } we can change the value as we go.

Why is this important?
It can be used to dynamically iterate through URLs (see next slide)
Or create other types of dynamic output programs like personalised emails.

Example of use: URL’s

```
for spec in speciality:
    looper = True
    while looper:
        page_num = page_num + 1
        url = 'https://www.doctoranytime.gr/s/{}?p={}'.format(spec,page_num)
        r=req.get(url)
        html= r.content
```

Exercise 1: Personalized Emails

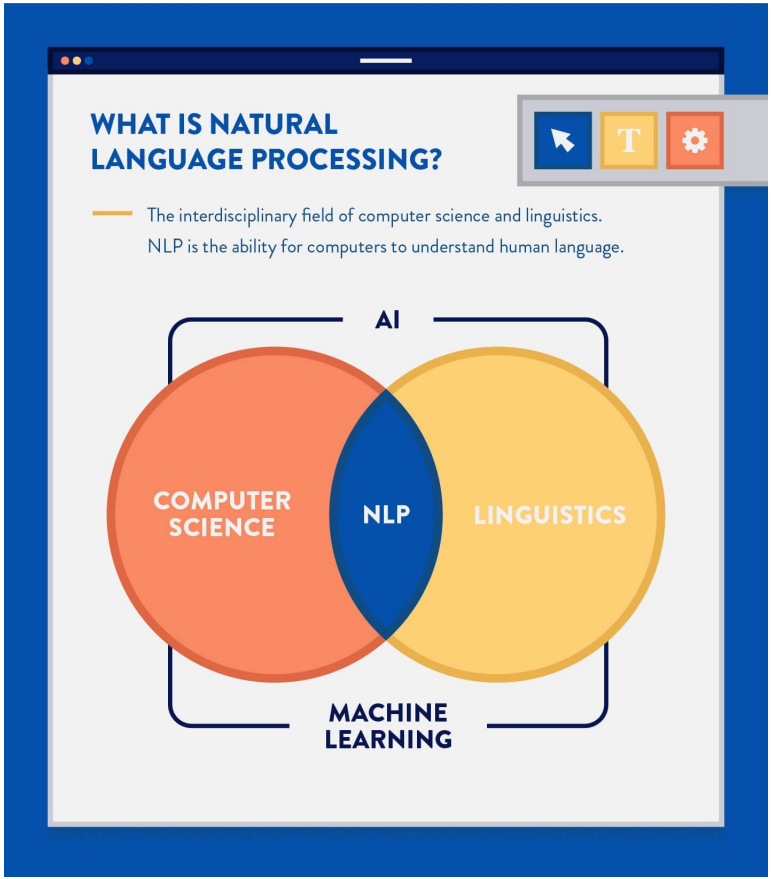
Create an array with names, using that array personalize a string (have their first name after a greeting). Then using the following set of commands, save the string into a file. Please keep the array to less than 5 names for the sake of brevity. Make sure to save each file to be like: the_name.txt

Allowed Time: 7 Minutes
Expected Length: 5 lines

Natural Language Processing

NLP is a computer science discipline which is based in converting (qualitative) language data into quantitative data. By using NLP we can create very powerful programs that can give insight to problems.

An example would be an analysis of the objectivity of an article.



Tokenization

NLP Analysis is almost often based on working with words one by one. This process is called Tokenization. Further Tokenization can also be done such that all words change to the same tense so the can be identified.

However we will only be using tokenization to create lists of words from text files.

Exercise 2: List of words

From the text for exercise 2. Create code that separates the words and creates an array of strings with all the words in the text.

This text only features “.” as punctuation not any other special chars. Don’t forget to use the forensics (visual) features of Jupyter when creating your code.

Allowed Time: 5 Minutes
Expected Length: 6 lines

Regular Expressions

A *regular expression* is a UNIX programming language that we can use within Python to manipulate UTF characters using a specialized syntax held in a pattern. Regular expressions are widely used in UNIX world.

We often call them Regexes.

Why do we need them? Say we are looking to analyse a snippet of text. Regexes allow us to understand a string and match it, split it and work with it in very powerful ways.

The *re* library

We can import the regular expression library by using the following command:

Import re

This library has a unique set of functions that we can use to work on some interesting string related projects. Today we will only discuss one use, however, I have emailed you a set of links where you can find more about regular expressions.

They are they type of thing you learn on the job since it is highly sophisticated and case specific.

Regex Sets

The essence of this “language” is the that we have a syntax that allows us to match with certain sets of characters. For example with all decimal digits or all non-alphanumeric characters.

These sets can be used to clean up qualitative string data before we analyze it.

‘[x\W]’

re.split()

In many cases we are interested in automating process where we collect specific data fields from a string. Don’t forget that with “string” we can refer to anything from one character to the entire wikipedia doc.

re.split(string, [maxsplit=0])

re.split(“\s”,str)

The above would split a string at the “white space” (ASCII \s) meaning the spaces

Exercise 3: Names (actual real life problem)

Using the text snippet I have emailed you, split the string at the '@' characters and the extract the instagram handles. You can do this by using a regex and then returning a cell in the array which split returns. Don't forget to save re.split() into a variable so you can call it.

Please inspect the array when splitted before you return anything to the user

```
In [15]: str = "hello@hello"
         re.split('\W',str)

Out[15]: ['hello', 'hello']
```

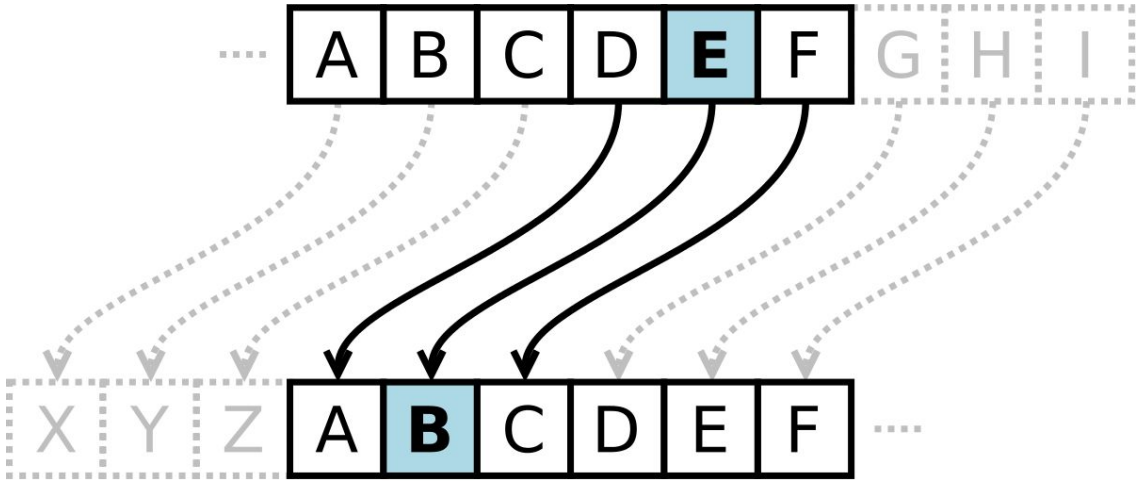
Character and Unicode

As we have said in the past, strings are comprised of Characters which are represented in the **unicode** UTF-8 encoding protocol. These characters are represented by a sort of a dictionary where a this → ‘a’ is represented by a number in the unicode table. When writing on a computer, the system understands the unicode convention of the characters and returns us the visual representation of the character. This compounded with the standardized application of UTF-8 allows us to text people around the world and correctly encode/visually represent characters.

Time to make a Caesar Cipher

History Recap

The Caesar Cipher is an ancient cryptographic technique. It is a single key symmetric encryption method. It is classified as a “substitution cipher” meaning that letters are substituted using a logic convention. Using a *key* the message is encrypted by shifting the letters to the letter which is: letter + key.



The catch: Modulo Arithmetic and Circles

However, for coding this, there is a catch. The english language, has 26 letters but Unicode has a few million. Meaning that when for example we use Key = 1 on ‘z’ we want it to return ‘a’ not whatever Z+1 is.

To do this we use **Modulo Arithmetic**

Modulo Arithmetic is a mathematical concept based on the remainders of a division.

$$10 \bmod 10 = 0$$

How does it work for us?

Modulo is cyclical.

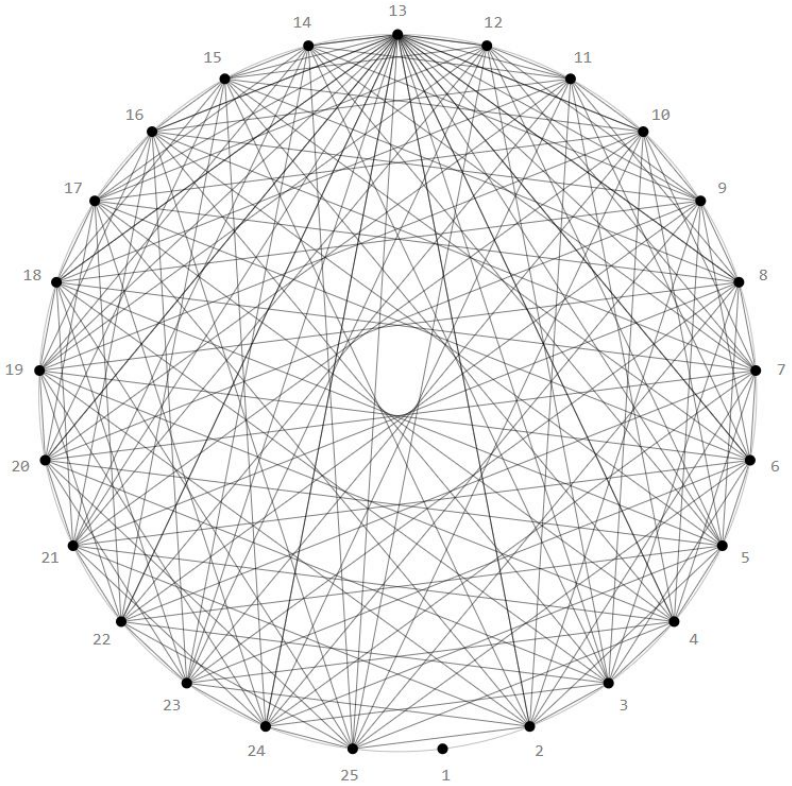
$$Given\ that\ n\ \in \mathbb{N}$$

$$x\ mod\ y = nx\ mod\ y$$

Remember that the remainder of a division can't be larger than the divisor (in this case *y*). This means that the domain of the function: nx mod y = [0,y]. This is exactly what we are looking for!

Also we should note that a key of 28 is the same as 2 which module also helps us with.

Modulo Patterns



Caesar Cipher Part 1: Dissecting the Problem

What are we trying to do?

- Get the **Plaintext** *input* from the user, as well as the **Key**
- *Slice* the string into characters
- Keep the **key** limited (use *Modulo*)
- Substitute each character of the **plaintext**
- Concatenate the array of now **Ciphertext** characters
- *Output* this **Ciphertext** to the user

Caesar Cipher Part 2: Inner Workings

We will cover, bullet points 2 & 4.

We can split a string into it's characters by using the function: *list(our_string)*
Which basically creates a **list** this can be used for many other things too.

Catch: Before we create our list we must change all characters to lowercase such that we have a single type. This is important since unicode has different indexes for upper and lowercase.

We can then use the: ord() function to change our characters. This converts a character into its numeric counterpart (int) we then add the **key** and convert back to char()

Extra Part 2: Run-off

At the end of our ord() expression we need to add the following:

(ord(char)+key-97**)**%26+97)

This weird piece of syntax just ensures that our characters remain within the Unicode lowercase constraints. 97 is the series pointer of ‘a’ and we just use %26 to cycle through the numbers.

This is a case specific snippet just for the C.Cipher and it would have been normal to end up having to look it up.

Caesar Cipher Part 3: The outter box

We will cover points 3 and 5.

We can do Modulo arithmetic using the ‘%’ operator. The expression:

$$Key = Key\%26$$

We can also concatenate the array by using the *join()* function which can be used to power-concatenate an array of strings into a single string.

Caesar Cipher Part 4: File I/O

We could make a program that takes user input using the *input()* function but we should make a better program. One that takes File Input and returns output as a file.

We can use the functions we have covered before to get input and output.

For those not present, the code is covered in the .py file.

Caesar Cipher Part 5: Optimization

We can optimize our code for large text blocks that have new lines. Remember that when encoding a .txt file with a space, we get a return of ‘\n’. We should clean-up our file after Input so we can accomodate for this.

To do this, we can use the *replace()* function to find ‘\n’ and replace it with ‘ ’ and due to the nature of *list()* we will just get a few extra empty spaces that can’t be modified.

Caesar Cipher Part 6: The final catch

Notice that when encoding spaces using our program we get special characters. This is a problem! To solve this we can just ensure that the substitution happens only

If main_list[i] != “ ”

Caesar Cipher Part 7: User Interface

Before we finish with the program we should add some U/I so we can make the user understand the program better. If you are following along online, you can make whatever type of U/I you want but an example is in the .py file for the problem.

Congrats you lot have made your first small scale deployable program!