Programming Abstractions in Python

CSE30 UCSC

INSTRUCTOR: DR. LARISSA MUNISHKINA

LECTURE: STRINGS AND RE

Agenda of Lecture 2:

1. Strings
2. Regular Expressions

Why Do We Study Strings?

Artificial Intelligence & Machine Learning

Data Preprocessing: Many ML models rely on clean, structured input. Textual data (strings) must be tokenized, normalized, and vectorized.

Feature Extraction: Strings are transformed into features using techniques like TF-IDF, word embeddings, or one-hot encoding.

Model Interpretation: Outputs like labels, summaries, or explanations are often strings that need to be parsed and presented clearly.

Natural Language Processing (NLP)

Core Data Type: NLP is fundamentally about understanding and generating human language — which is all strings.

String Manipulation: Tokenization, stemming, lemmatization, and parsing are all string-heavy operations.

Cybersecurity

Input Validation: Preventing injection attacks (SQL, XSS) requires careful string sanitization.

Encryption & Hashing: Strings are often the input/output of cryptographic functions.

Log Analysis: Security tools parse string-based logs to detect anomalies or breaches.

Compilers and Programming Languages

Lexical Analysis: Compilers break source code (strings) into tokens.

Syntax Parsing: Strings are parsed into abstract syntax trees (ASTs).

Error Reporting: Compiler diagnostics are string-based and must be clear and informative.

Part I: Strings

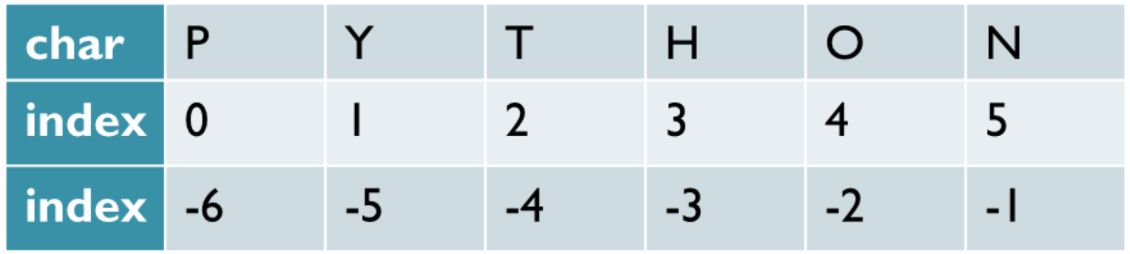
Strings

String is a very important data type in Python and in programming in general.

Strings are used to represent text, meaning they can represent any information.

Code is written as text, so it can be represented as a string.

A string is an array of characters:



char	P	Y	T	H	O	N
index	0	1	2	3	4	5
index	-6	-5	-4	-3	-2	-1

String Manipulation

Since a string is an array of characters, it is a sequence (ordered) and can be manipulated as regular arrays - so, we can use the following operations:

- indexing
- slicing

```
>>> s = '123456'
>>> s[1]
'2'
>>> s[::2]
'135'
```

String Concatenation

Since strings are immutable, we cannot insert or delete characters.

However, we can concatenate strings and use slicing to omit characters within a string:

```
>>> s = '123456'

>>> s[ : : 3] + s[1]

'142'
```

We can concatenate strings from a list of strings using the method **join**:

```
>>> ' '.join(['a', 'b', 'c'])

'a b c'
```

String Splitting

A string can be split into parts using a delimiter.

By default, the delimiter is a space character.

```
>>> s = 'How are you?'
>>> s.split()              # the delimiter is '  '
['How', 'are', 'you?']
>>> s.split('are')         # the delimiter is 'are'
['How ', ' you?']
```

Part II: Regular Expressions

Regular Expressions (RE)

Purpose:

- Create a notation system for patterns
- Search for patterns in strings
- Quickly extract information from text data

Definition:

- RE is a tool consisted of a **notation system** for string patterns and **operations** to manipulate patterns

Examples:

- [ ] denote a set of characters
- [A-Z] – means all uppercase letters from A to Z
- . – means any character
- \d – means any digit [0-9]

Regular Expressions

Char	Description	Example	Char	Description	Example
\char	escape seq.	‘*’ means ‘*’	.	any char	a. matches ab
\d	[0 – 9]	‘8’	*	>= 0 times	ba* matches b
\D	[^0 – 9]	‘A’	+	>=1 times	ba+ matches baa
\w	[A-Za-z0-9_]	‘h’	?	0 or 1 time	ba? matches ba
\W	[^A-Za-z0-9_]	‘&’	^	start of line	^and
\s	[\n\t\f]	space in ‘a b’	$	end of line	and$
\S	[^\n\t\f]	‘!’ in ‘yes!’	[^]	not	[^xyz]
\b	boundary	‘f’ in ‘it is fun’	[|]	or	[A|E]
\B	not boundary	‘u’ in ‘it is fun’	{m}	>= m times	A{2}
(…)	grouping	([A-Z])([0-9])	{m, n}	>= m <= n times	A{2,3}

Using Regular Expressions

Use raw-string for RE

r ‘^$’ - matches all empty lines

Use **re** module and **compile** method

```
import re
pattern = re.compile(r’\d+’)
matches = pattern.findall(“200 students and 10 teachers”)
print (matches)
>>>[200, 10]
```

RE Methods

RE compile method	• compile() – creates an RE pattern object using an RE string
RE Pattern Object	• match() – finds a match from the beginning • search() – finds a match within text • findall() – finds multiple matches within text • sub() – substitutes a pattern with a string
RE Matched Object	• start() – start of the match • end() – end of the match • span() – the range of the match • group() – the matched string

Design a pattern to match all valid identifiers

(we need to check later if a name is not a keyword)

^[A-Za-z_][A-Za-z_0-9]*$

^[A-Za-z_]\w*$

Test if a string is a valid identifier

```
import re
pattern = re.compile ( r'^[A-Za-z_][A-Za-z_0-9]*$' )
while True :
    identifier = input("Enter an identifier: ")
    match = pattern.match(identifier)
    if match :
        print ("It is a valid identifier.")
    else :
        print ("It is not a valid identifier.")
    ans = input ("Do you want to continue? [Y/N] ").lower()
    if ans != 'y' :
        break
```

Regular Expressions for Passwords

```
# verify a valid password
import re
p_upper = re.compile(r’[A-Z]')
p_lower = re.compile(r’[a-z]')
p_digit = re.compile(r'[0-9]')
p_symbol = re.compile(r'[@#$%&?! ]')
while True :
    password = input("Enter a password: ")
    m_lower = p_lower.search(password)
    m_upper = p_upper.search(password)
    m_digit = p_digit.search(password)
    m_symbol = p_symbol.search(password)
```

```
# the code continues here
    if not m_lower :
        print ("You need to use a lowercase letter.")
        continue
    if not m_upper :
        print ("You need to use an uppercase letter.")
        continue
    if not m_digit :
        print ("You need to use a digit.")
        continue
    if not m_symbol :
        print ("You need to use a special character.")
        continue
    break
```

Self-testing (6 min)

Write a RE that match any phone numbers using the following format (xxx)xxx – xxxx where x is a digit.

Write a RE that match any phone numbers using the same format and (831) area code.

Write a program that finds phones in text.

Let’s check your answers!

Write a RE that matches any phone number using the following format (xxx)xxx – xxxx where x is a digit.

- \([0-9][0-9][0-9]\) [0-9][0-9][0-9] – [0-9][0-9][0-9][0-9]
- \(\d\d\d\) \d\d\d – \d\d\d\d
- \(\d{3}\) \d{3} – \d{4}

Write a RE that matches any phone number using the same format and area code 831.

- \(831\) [0-9][0-9][0-9] – [0-9][0-9][0-9][0-9]
- \(831\) \d\d\d – \d\d\d\d
- \(831\) \d{3} – \d{4}

Let’s check your answers!

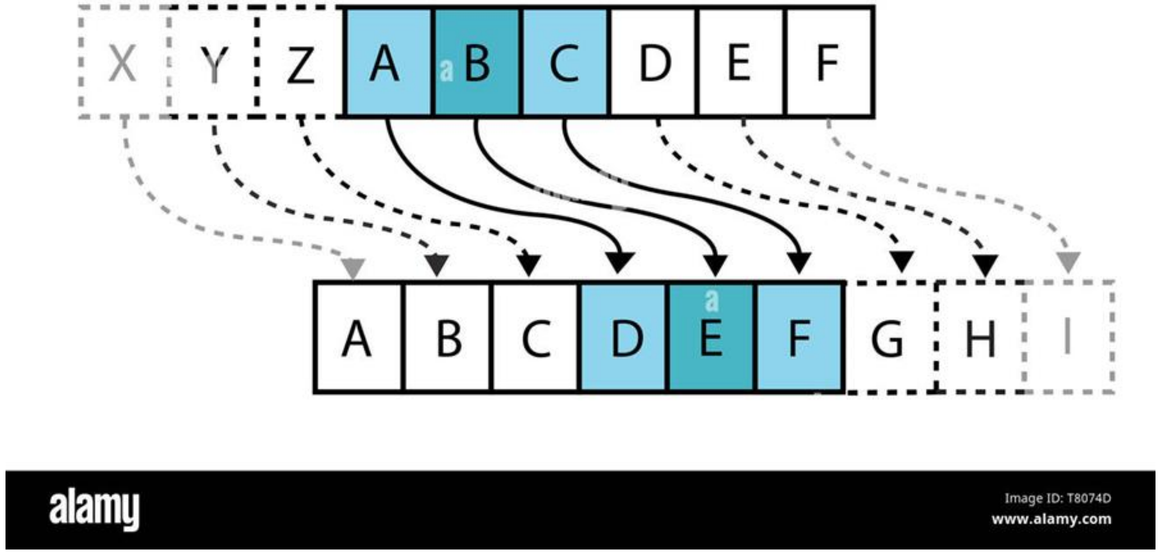
```
import re
p = re.compile(r'\(?\d{3}\)?-?\d{3}-?\d{4}')
while True :
    text = input("Enter text with phone numbers: ")
    m1 = p.search(text)
    if m1 :
        print ("Search group: ", m1.group())
    else :
        print ("Search: no number")
        continue
```

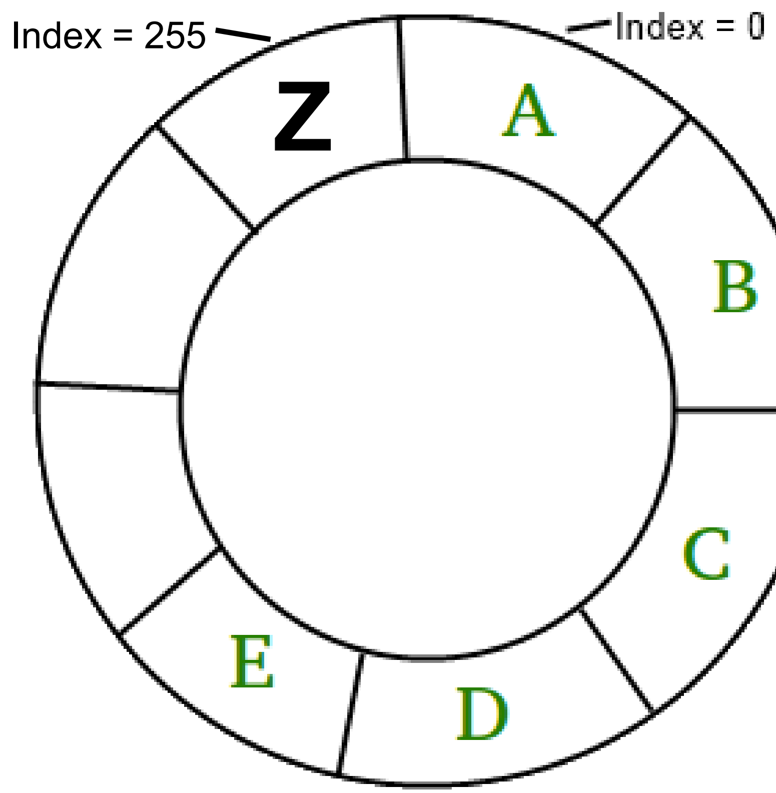
```
m2 = p.findall(text)
print ("Findall: ", m2)
for m in m2 :
    print ("Findall group: ", m)
m3 = p.match(text)
if m3 :
    print ("Match: ", m3.group())
else :
    print ("Match: no exact match")
break
```

Caesar Cipher

Caesar Cypher

```
def encode(self, text):
        data = ''
        # your code goes here
        return data
def decode(self, data):
        text = ''
        # your code goes here
        return text
```





Circular representation of an array

Caesar Cypher

1. To apply a Caesar cipher, we need to use a circular array.
2. A circular array is an array in which the last element is next to the first element.
3. Indexes of a circular array are calculated using the modulo operation:

   index = (alphabet.index(character) + shift) % 26

   index is in the range from 0 to 25

Homework & Assignments

- Read NB 1 and optionally Alfaro’s Ch. 1 and 2 (Intro and Data Structures)
- NB 1 is due on Monday, October 6th
- (Optional) read Ch. 1.1-1.17 in the ‘Problem Solving’ textbook