REGEX

ביטויים רגולריים (regular expression או בקיצור regex) מאפשרים למתכנת למצוא תבניות של תווים בקוד במקום לבצע שאילתות מורכבות על הטקסט.
הביטוי הרגולרי של שפות תכנות בנוי על בסיס ההגדרה המתמטית של ביטוי רגולרי.
אז למה צריך בכלל ביטויים רגולרים?
נסתכל על הדוגמא הבאה- נניח אנחנו רוצים לבנות פונקציה שבודקת האם מחרוזת מסויימת מייצגת מספר או לא.
אם ננסה לבנות פונקציה ייעודית פשוטה ואינטואיטיבית זה ייקח לנו הרבה מאוד שורות קוד, מספר ישראלי מוגדר, נכון לכתבית שורות אלה, בצורה 05x-xxx-xxxx ובמקום ה-x מספרים ב- [1,9]:

```
def is_a_phone_number(text:str)->bool:
    '''Checks if a string is a valid  phone number '''
    # 05x-xxx-xxxx x ∈ [1,9]
    if len(text) != 12:
        return False
    if text[0:2] != '05':
        return False
    nums = text.split('-')
    if len(nums) != 3:
        return False
    for num in nums:
        if not num.isdecimal():
            return False
        if num == nums[2] and len(num)!= 4:
            return False
    return True
```

והעסק יכול להסתבך אף יותר, נניח אנחנו רוצים למצוא את כל המספרי טלפון שנמצאים במחרוזת כלשהי, הגישה הפשוטה ביותר תיהיה לבצע חיפוש שלם:

```
from typing import List
def find_phone_numbers(text:str)-> List[str]:
    '''Finds all phone number in a string '''
    list_of_phone_numbers = list()
    for i in range(len(text)):
        if is_a_phone_number(text[i:i+12]):
           list_of_phone_numbers.append(text[i:i+12])
    return list_of_phone_numbers
```

סהכ כל התהליך לקח לנו כ-20 שורות קוד, אבל אם היינו יכולים לקצר את התהליך ולהוריד אותו לשלוש שורות קוד בלבד? בשביל זה נצטרך להשתמש בביטויים רגולרים.
המודול של regex נמצא כבר בספרייה הסטנדרטית של פייתון ואפשר להשתמש בו ללא צורך בהתקנה של תוספים זרים.
בשביל להשתמש בביטויים רגולרים נצטרך לייבא את המודול re ולו יש כמה פונקציות חשובות.
הראשונה היא הפונקציה ()compile שמקבלת איזשהו ביטוי רגולרי ומחזירה אובייקט מטיפוס pattern (תבנית), עם האובייקט הזה אפשר לבצע שאילתות על טקסט מסוים כמו למשל למצוא את כל המופעים של מבנה הנ"ל או מופע אחד

שלו וכו'.
לפני שניכנס לפרטים בנוגע לשימוש ב-regex, נראה איך היינו פותרים את הבעיה:

```
from typing import List
def find_phone_numbers_regex(text:str)-> List[str]:
    import re
    phone_pattern = re.compile(r'05\d-\d\d\d-\d\d\d\d')
    phones = phone_pattern.findall(text)
    return phones
```

לא נראה שינוי מהותי, אבל תחשבו שעכשיו לא צריך בכלל את הפונקציה is_a_phone_number() , כך שזה חסך מעל לעשר שורות קוד.

משתנים-

כפי שאמרנו קודם ביטויים רגולרים מבוססים על תחבירים מתמטים פשוטים.
בתחביר רגולרי צריך להגדיר את סוג הטיפוסים השונים שעליהם התחביר בנוי, הטיפוסים יכולים להיות מספריים או אותיות, למשל ראינו בדוגמא לעיל עם מספרי הטלפון טיפוס של מספרים: '\d' שימו לב שעבור ביטוי מדויק שאנחנו רוצים למצוא אנחנו ממש כותבים אותו, למשל בדוגמא כשרצינו לבדוק שבביטוי יש את התו '-' ממש כתבנו אותו בשאילתה.
בשביל למצוא איזשהן אותיות לטיניות (באל"ף בי"ת האנגלי הסטנדרטי) או מספרים ותווים שכחים נשתמש ב-'\w' , למשל אם נרצה לבדוק האם ביטוי מכיל אות לטנית סטנדרטית (לפחות אחת).
קבוצה הפוכה לשני הטיפוסים יהיו אותן האותיות רק באותיות גדולה, למשל נניח אנחנו רוצים למצוא את הביטוי הראשון בטקסט בלי אותיות לטיניות או מספרים ותווים מיוחדים נחפש search('\W'), או הביטוי הראשון שאין בו מספרים search(r'\D')
רווח הוא גם נחשב תו שכיח בטקסט ואותו מגדירים ב- '\s' או בלי רווח '\S' .

חוץ מזה יש כמה סימונים שעוזרים לנו למצוא טקסט לפי מיקום החיפוש בטקסט, כך נוכל לראות מופעים שנראים בתחילת הטקסט, בסופו, או במיקום אחר:
'\A' – מאתר אם החיפוש מופיע בתחילת הטקסט, למשל search(r'\AHead') בודק האם הביטוי 'Head' נמצא בראש הטקסט.
'\B'- בודק האם הביטוי נמצא אבל **לא** בתחילת הטקסט למשל search(r'\Bmiddle') או לא בסופו: search('middle\B').
'\b' – ההפך מ-'\B', בודק האם הביטוי נמצא בתחילת הטקסט (search(r'\bHead')) או בסופה(search(r'end\b'))
'\Z'- בודק האם הביטוי נמצא בסוף הטקסט search(r'end\Z').

לפני שנמשיך, חלקכם בטח שאלתם מה זה ה-r הזה בתחילת המחרוזת, אז ה-r מסמל מחרוזת מיוחדת מסוג raw string, זאתי מחרוזת שמתעלמת מ-'\' ומתייחסת אליו כאילו הוא סוג של תו, כלומר במקום שנכתוב '\\' נוכל לכתוב רק פעם אחת עם ציון שזאתי מחרוזת r לפני.

קבוצות –

אפשר להגדיר קבוצת תווים יחד באותה השאילתה ע"י שימוש ב-'[ ]', וכל מה שיכנס לסוגריים יחשב כחלק מהקבוצה, למשל אם נרצה למצוא את המופע הראשון של האותיות a,b או האותיות הגדולות שלהן נשתמש ב-search(r'[abAB]') וזה יחזיר לנו את המופע הראשון שבו מופיעה אחת מהאותיות שציינו.
גם טווח אפשר להגדיר עם התו- '-' בתוך הסוגריים, למשל בין האותיות a ל-k יהיה [a-k] או המספרים בין אחד ל-7 זה [1-7] וכו'.
שימו לב שהקבוצה היא קבוצת תווים ולא מילים שלמות.
אם נרצה לשלול קבוצה נשתמש באופרטור ^ בתחילת המילה, למשל כדי למצוא מתי הפעם הראשונה בביטוי בה לא מופיע אחד מהתווים b,a או c : search(r'[^abc]').

אופרטורים –

יש כמה אופרטורים לשאילתות:
'+' – מופע אחד או יותר של ביטוי כלשהו, למשל המילה no או ...nooo נחפש search(r'no+')
'*' – אפס או יותר מופעים של התו: למשל אם מופיע הביטוי ...yess או yes או רק ye : search('yes*').
'?' – מופע אחד או אפס של התו- למשל dog או dogs נחפש search('dogs?')
'{}' – בודק אם בטקסט יש ביטוי מסויים כשתו (או מילה) בסיומו מלווה מספר פעמים מוגדר, למשל בשביל למצוא את המילה all נוכל להשתמש בשאילתה search(r'al{2}') שמראה ש-l אמור להופיע פעמיים בדיוק, ואפשר גם להגדיר טווח של מספרים למשל בין 2 ל-5 כך search(r'al{2,5}').
הערה: החיפוש של הפונקציה הוא חיפוש חמדני שמחפש אתת המקסימום בתוך הטווח, למשל נרצה למצוא בין שלוש לחמשיה מספרים בתוך מחרוזת של עשרה מספרים, החיפוש יחזיר לנו חמישה מספרים ולא שלושה.
אם רוצים חיפוש לא חמדני צריך להוסיף ? לאחר הסוגריים המסולסלים, במקרה זה סימן שאלה לא מסמן אפס או אחד , אלא חמדני או לא חמדני.
'.'- מחליף כל תו למעט '\n' ('\w' מכיל רק מספרים, אותיות לטיניות סטנדרטיות ו-'_' , אבל לא תווים כמו '\n' וכדו').
למשל search(' he..o') יביא כל מילה או ביטוי שעטוף באותיות he ונגמרות ב-o.
'^' - מתחיל עם, כמו '\A' למשל אם הטקסט מתחיל עם המילה hello : search('^hello')
'$'- נגמר עם, כמו '\Z' למשל נגמר עם המילה hello : search('hello$')
'|' – אופרטור או, בשביל לברור בין כמה אפשרויות למשל: yes או no נעשה search('yes|no').
'()' – לכידה לקבוצה אחת, כדי להחשיב כמה ביטויים יחד בקבוצה, שימושי בעיקר כשרוצים לפרק לחלקים את הביטוי אותו אנחנו רוצים למצוא, אם נחזור לדוגמא הראשונה שלנו עם מספרי הטלפון, יכולנו לפרק את הקוד לנניח קידומת והמספר עצמו, למשל: search(r'(\d\d\d)-(\d\d\d-\d\d\d\d)'), סתם ככה לא נראה איזשהו שינוי בקוד אם נשתמש בפונקציה group(), אבל עכשיו נוכל להוסיף לה פרמטר שיגדיר איזו קבוצה נרצה, למשל בשביל לקבל רק את הקבוצה הראשונה נוסיף ארגומנט עם הסיפרה אחת: group(1) זה יחזיר רק את הקבוצה הראשונה שנתפסה, במקרה זה הקידומת של המספר, או אם נבצע findall זה יחלק כל מציאה ל-tuple לפי הקבוצות שהגדרנו:

```
from typing import List
def find_phone_numbers_regex(text:str)-> List[str]:
    import re
    phone_pattern = re.compile(r'(05\d)-(\d\d\d-\d\d\d\d)')
    phones = phone_pattern.findall(text)
    return phones

txt = '054-444-4444 gj fdkl055-555-5555mvkdl053-545-4545'
find_phone_numbers_regex(txt)
```

```
[('054', '444-4444'), ('055', '555-5555'), ('053', '545-4545')]
```

הערה: אם רוצים למצוא ממש את הערך של אחד מהסימונים המיוחדים אפשר להוסיף '\' לפני התו, למשל כדי למצוא '?' נכתוב '\?'.

פונקציות –

בביטויים רגולרים יש כמה פונקציות שכיחות:
הפונקציה compile() כפי שראינו מביאה לנו תבנית של ביטוי רגולרי , אבל אין חובה לקמפל כל פעם ביטוי ואפשר לבצע כל פונקציה שאנחנו מבצעים עם אובייקט תבנית ישירות, רק צריך להגדיר את הטקסט עליו אנחנו עובדים כארגומנט לפונקציה. למשל שתי הדרכים הבאות שוות:

```
pattern = re.compile(r'(\d\d\d)-(\d\d\d-\d\d\d\d)')
pattern_groups = pattern.search(txt)
print(f'pattern {pattern_groups.group()}')

direct_groups = re.search(r'(\d\d\d)-(\d\d\d-\d\d\d\d)' , txt)
print(f'direct {direct_groups.group()}')
```

לפונקציה compile אפשר להוסיף כמה סוגים של פרמטרים(flags) כדי להגדיר את סוג המחרוזת של התבנית שעליה עובדים.

re.A או re.ASCIIמגדירה שהמחרוזת מתיחסת רק סינטקס של ascii קוד ולא Unicode (שהוא הברירת מחדל).

re.I או re.IGNORECASE מתיחסת לאותיות גדולות ולאותיות גדולות בצורה שווה:

```
all_cases_a_patt = re.compile('a', re.IGNORECASE)
all_cases_a = all_cases_a_patt.findall('And they have all they wanted AMEN')
print(all_cases_a)
```

```
['A', 'a', 'a', 'a', 'A']
```

re.S או re.DOTALL מגדיר שהתו – '.' יוכל לייצג כל אות ממש כולל ירידת שורה.

re.X או re.VERBOSE מתעלם מירידת שורה בקריאת התבנית, ובכך מאפשר לכתוב טקסט יותר קריא:

```
phone_patt = re.compile(R'''
05\d-
\d{3}-
\d{4}
''', re.VERBOSE)
phones = phone_patt.findall('054-444-4444 gj fdkl055-555-5555mvkdl053-545-4545')
print(phones)
```

```
['054-444-4444', '055-555-5555', '053-545-4545']
```

כדי לשלב להעביר כמה flags כאלה כארגומנט לפונקציה צריך להשתמש באופרטור ה-bitwaise '|', כן זה קצת משונה אבל ככה מעבירים אותם.
למשל כדי להתעלם מאותיות גדולות, גם שנוכל להגדיר כל משתנה עבור '.' וגם שנתייחס רק ל-ascii קוד, והתבנית שלנו היא '.+' נכתוב compile('.+',re.A|re.S|re.IGNORECASE) :

```
find_all_patt = re.compile('.+',re.A|re.S|re.IGNORECASE)
find_all = find_all_patt.search('every thing I write \nwill be found')
print(find_all.group())
```

```
every thing I write
will be found
```

באובייקט החוזר מהפונקציה search(), חוץ מ-group אפשר לקבל גם את האינדקס של תחילת המופע הראשון וסופו עם הפונקציה span(), כמו כן כל שאילתה שאנחנו מבצעים שומרת גם את המחרוזת עליה היא פועלת במשתנה string (אם הוא לא None) :

```
txt = '054-444-4444 gj fdkl055-555-5555mvkdl053-545-4545'
span = re.search(r'(\d\d\d)-(\d\d\d-\d\d\d\d)' , txt)
print(span.span())
print(span.string)
```

```
(0, 12)
054-444-4444 gj fdkl055-555-5555mvkdl053-545-4545
```

הפונקציה split() דומה לפונקציה split() של מחרוזות רק שמאפשרת לבצע את הפיצול עם regex .

```
phone_number_row = 'Tom Pythonovitz, 055-555-5555 Tammi Pythonovitz, 054-444-4444'
list_of_users = re.split(r', \d\d\d-\d\d\d-\d\d\d\d',phone_number_row)
print(list_of_users)
```

```
['Tom Pythonovitz', ' Tammi Pythonovitz', '']
```

עוד פונקציה שיכולה לשנות את המחרוזת היא הפונקציה sub שמשנה את השאילתה למחרוזת מוגדרת, למשל להחליף בין רווחים ל-$, ואפשר גם להוסיף פרמטר לכמה מופעים יתרחש השינוי:

```
txt = "The rain in Spain"
x = re.sub("\s", "9", txt)
print(x)

y = re.sub("\s", "9", txt, 2)
print(y)
```

```
The9rain9in9Spain
The9rain9in Spain
```

כל קבוצה מהתבנית ממוספרת לפני המופע שלה, נניח התבנית שלנו היא כזו: '(\w) (\d) (\.)+'
אז הקבוצה הראשונה היא '(\w)' השנייה היא ' (\d)' והשלישית '(\.)+' .
אפשר לגשת לכל אחת מהקבוצות בתבנית עם \ ומספר הקבוצה, למשל כדי לגשת לקבוצה אחת נכתוב '\1'.
נראה דוגמא: בשב"כ החליטו לבנות פונקציה שמצנזרת את שמות הסוכנים שלה כך שבמקום שיופיע שמם תופיע רק האות הראשונה של שמם:

```
import re

txt = "Agent Adam is going to meet Agent Yosi in the same location they met last time"
agent_pattern = re.compile("Agent (\w)\w+")
x = agent_pattern.sub(r"Agent \1", txt )
```

```
print(x)
```

```
Agent A is going to meet Agent Y in the same location they met last time
```

הפונקציה ()findall מוצאת את כל המופעים שאינם חופפים בטקסט.
הפונקציה מחפשת משאל לימין , ומחזירה ,בסדר בו היא מצאה, רשימה של מחרוזות שקשורות לתבנית מהטקסט.
בשונה מ-search הפונקציה מחפשת לפי קבוצות, כך שאם התבנית שהתקבלה מכילה קבוצה, הפונקציה תמצא את הקבוצה ולא את כלל המחרוזת, למשל בחיפוש אחר 'Bat(wo)?man' הפונקציה תחפש אחר הקבוצה wo ואם תמצא תחזיר את צמד האותיות ולא את כל המשפט, ומשום שכתבנו את זה עם '?' היא תחזיר גם מחרוזת ריקה.
בשביל לפתור את זה חלקית מומלץ להכניס את כל הביטוי לקבוצה או להשתמש ב- '|' (אופרטור או):

```
bat_gender = re.findall('Bat(wo)?man', 'Batman VS Superman')
print(bat_gender)
bat_gender = re.findall('Bat(wo)?man', 'Batwoman VS Superman')
print(bat_gender)
bat_gender = re.findall('(Bat(wo)?man)', 'Batman VS Superman')
print(bat_gender)
```

```
['']
['wo']
[('Batman', '')]
```