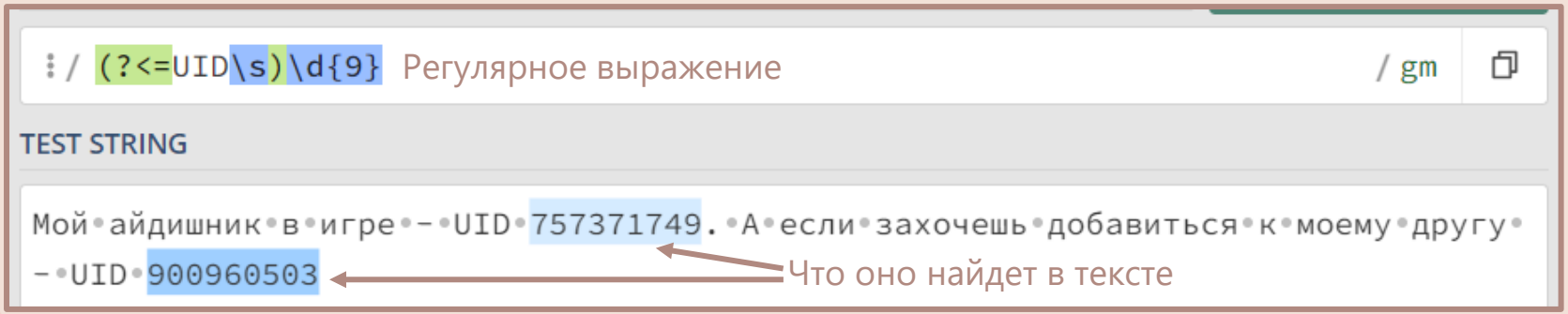
Программирование в лингвистике

Регулярные выражения

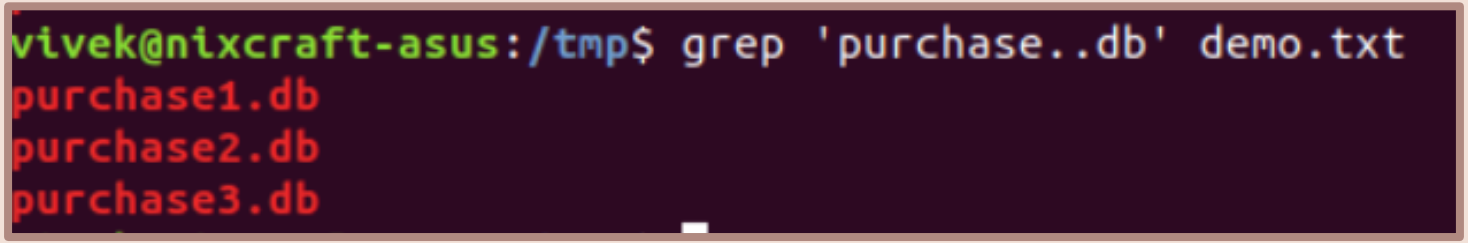
Регулярные выражения

Регулярное выражение - это шаблон, который соотносится с последовательностью символов в тексте. По существу, РВ - это обобщенная запись для разных вариантов строк, например, можно обобщить, как выглядят все телефонные номера в тексте или все e-mail адреса. РВ придумали в 1950-х гг., когда математик Стефен Коул Клини формализовал описание регулярного языка.



grep

Одно из первых применений РВ - это поиск по текстам, содержащимся внутри файлов. До 90-х годов прошлого века операционные системы не имели графического интерфейса, а во многих файлах лежала текстовая информация. В 70-з гг. программисты создали программу grep, которая с помощью РВ ищет все файлы, внутри которых есть совпадающие тексты. grep работает из командной строки в системах unix (linux, MacOS) и до сих пор активно используется при работе с серверами.



Типы синтаксиса РВ

У РВ есть несколько типов синтаксиса (flavors), которые различаются функциональностью. Самые известные такие:

- POSIX (имеет два варианта: базовый и расширенный, BRE & ERE)
- Perl 5 (разработчики языка Perl добавили свои плюшки)
- Perl 6 = Raku Rules
- PCRE (Perl Compatible Regular Expressions)

Некоторые ЯП имеют свои версии синтаксиса РВ, в основном очень похожие на PCRE; Python в их числе.

Сайты для тестирования РВ

Существует множество сайтов, которые предназначения для тестирования шаблонов регулярных выражений. Сами РВ обычно используются в программах для того, чтобы что-то найти в обрабатываемом тексте или проверить, удовлетворяет ли строка условиям, но прежде чем запускать готовую большую программу, шаблон всегда лучше потестировать.

- https://regex101.com/ - очень удобный сайт для тестирования
- https://pythex.org/ - сайт с РВ исключительно для версии питона
- https://www.regexpal.com/ - сайт с РВ для JavaScript, для простых случаев его тоже можно использовать

Для желающих попрактиковаться есть https://regexcrossword.com/ - это сайт с кроссвордами-РВ, где вам нужно подбирать регулярные выражения, чтобы заполнить табличку.

Когда использовать РВ?

Если вы уверены, что без РВ не обойтись. Прежде чем писать РВ, подумайте, нельзя ли решить задачу стандартными методами строк или множеств? РВ - мощный инструмент, но для некоторых задач он чрезмерен.

У РВ есть своя вычислительная сложность: для компиляции регулярного выражения длины n компьютеру приходится тратить 2^n времени.

А еще в РВ легко запутаться, даже опытному программисту бывает не так-то просто составить правильный шаблон, который не захватывал бы лишнего и не пропускал бы нужного.

Для чего используют РВ?

Регулярные выражения часто используются для:

- поиска по корпусам (в ГИКРЯ поддерживаются некоторые возможности РВ);
- поиска по текстам внутри файлов (на серверах - но даже Windows кое-что знает про РВ, хотя у нее свои правила);
- токенизации, сегментации по предложениям;
- обработки путей файлов и т.д.

Правила составления шаблонов

Регулярное выражение читается слева направо и ищет посимвольно. То есть, допустим, мы хотим найти слово из трех букв. Мы можем представить его как

---	---	---

Дальше уже нужно подумать, что поставить на место каждого прочерка. Если мы хотим искать конкретные буквы, то можем прямо их и написать, например,

c	a	t

Найдет нам все вхождения "cat" в тексте (ну или хотя бы одно, в зависимости от функции).

Специальные последовательности

Мы такие вещи уже знаем из обычного питона: \n и \t. То есть, специальная последовательность - это какая-то буква (или набор их) после бэкслеша. РВ распознают и стандартные последовательности Python, но имеют и некоторые свои. Какие нам пригодятся:

\d	любая цифра (те символы, которые в юникоде имеют ярлык "digit")
\w	любая цифра или буква (те символы, которые в юникоде считаются цифробуквенными). Примечание: туда еще входит нижнее подчеркивание _
\s	любой пробельный символ (пробел, неразрывный пробел, табуляция, перенос на новую строку...)
\b	граница слова: это не символ, а условие, которое говорит, что на этом месте должно быть \W\w или наоборот

Метасимволы

Это такие символы, которые означают не сами себя, а что-то другое.

.	любой символ вообще (кроме пробельных символов)
|	оператор "или" (логическое объединение)
+, *, ?, {}	квантификаторы
^	начало строки
$	конец строки (И $, и ^ функционирует как \b)
()	группа
[]	класс

Если нужно искать сами эти символы, то их нужно экранировать с помощью бэкслеша: \.
позволит искать точку, например. Сам бэкслеш тоже, кстати, нужно экранировать: \\

Квантификаторы

Мы можем искать телефонные номера с помощью \d, например. Предположим, наши номера состоят из шести любых цифр, разделенных дефисами:

__	__	-	__	__	-	__	__

Мы можем записать такие номера РВ как:

\d	\d	-	\d	\d	-	\d	\d

Но если мы хотим одиннадцатизначные номера, в которых цифры просто идут подряд? Или все номера, где от 6 до 11 цифр?

С квантификаторами наш одиннадцатизначный номер превратится в **\d{11}**.

Квантификаторы

Итак, квантификаторы применяются к одному символу, стоящему слева от них.

{n}	найдет ровно n таких символов
{n,m}	найдет не меньше n и не больше m символов
{n,}	найдет не меньше n, но чем больше, тем лучше
{,m}	найдет от 0 до m символов
?	найдет либо один такой символ, либо ни одного
*****	найдет либо 0 таких символов, либо чем больше, чем лучше
+	найдет от 1 до бесконечности символов!

Квантификаторы

Таким образом:

\d{3}	найдет 3 цифры
\d{2,3}	найдет либо 2, либо 3 цифры, но охотнее 3
\d{2,}	найдет минимум 2 цифры, но лучше побольше
\d{,3}	найдет от 0 до 3 цифр
\d?	либо найдет пустоту, либо 1 цифру
\d*	либо найдет пустоту, либо как можно больше цифр
\d+	найдет 1 и больше цифр

Такие квантификаторы называются жадными: они ищут как можно больше повторяющихся символов. Можно сделать квантификатор ленивым, чтобы он искал поменьше, если поставить сразу после него ?, например, **\d+?**, если сможет, будет стараться обходиться только одной цифрой.

Классы символов

Квадратные скобки обозначают классы символов. Класс символов - это некоторый набор символов, которые нас устраивают в этом месте. Мы уже знаем некоторые классы: \d, \w, \s. Но можно задавать и свои, произвольные. Например, мы хотим найти в тексте все слова these, но those нас в принципе тоже устраивают. Тогда:

t	h	[eo]	s	e

То есть, внутри квадратных скобок можно просто перечислить все символы, которые нам нравятся. Сами квадратные скобки будут занимать только одну позицию: они обозначают один символ, любой из перечисленных внутри них.

Классы символов

Внутри квадратных скобок в основном метасимволы теряют свою силу, и '.' делается обычной точкой, например. Но некоторые ее, наоборот, приобретают или изменяют.

- **Дефис**, если не стоит с краю, обозначает **диапазон**: [0-9] - это все цифры от 0 до 9. Диапазон берется из таблицы юникода. Мы уже знаем, что в основном буквы там идут по алфавиту, но буква ё, например, убежала не туда. Значит, чтобы захватить весь русский алфавит с ё, нужно писать: [а-яё]. А если нас еще и заглавные буквы интересуют, то придется написать [А-ЯЁа-яё].

- ^ из начала строки превращается в **отрицание**, но только в начале класса. Если нас интересует все, кроме того, что мы перечислили, можно записать: [^а-яё]. Такая запись будет означать "все, кроме кириллических строчных букв".

- Не теряет свою силу только \, потому что внутри класса мы можем написать \d или \t, например.

Группы

Группы нужны в основном для каких-то технических целей. Самая, пожалуй, частая - это для того, чтобы ограничить действие оператора "или".
По умолчанию "или" работает на все РВ целиком: он считает, что нас устраивает или все, что стоит слева, или все, что стоит справа. То есть,

`a|bc` будет искать либо a, либо bc.

Но если мы хотим искать либо ac, либо bc? Ну, можно написать явно: ac|bc, но часто повторяющаяся часть оказывается слишком длинной. Тут на помощь нам приходят скобки, которые выделят символы в группу и ограничат |:

`(a|b)c = ac|bc`

Группы

Группы также можно использовать для того, чтобы ссылаться на них. Группы в РВ нумеруются (по левой скобке), и к любой можно обратиться по ее номеру:

`(\w+) так \1` найдет все выражения вида X так X

Иногда, правда, группа нам нужна не для того, чтобы на нее ссылаться, и тогда используются группы без захвата содержимого:

`(?:xxx)`

Проверки

Проверка - это когда мы хотим проверить какое-то условие, чтобы слева или справа от нашего искомого текста были определенные вещи, при этом включать их в результат поиска не хотим.

Проверки бывают четырех видов:

(?=...)	опережающая позитивная (positive lookahead)	смотрим справа от того, что ищем
(?!...)	опережающая негативная (negative lookahead)	
(?<=...)	ретроспективная позитивная (positive lookbehind)	смотрим слева от того, что ищем
(?<!...)	ретроспективная негативная (negative lookbehind)	

Проверки

Например, мы хотим найти в тексте имя "Вася", но только такое, чтобы после него шла фамилия "Пупкин". Тогда нам нужна опережающая позитивная проверка:

```
Вася (?=Пупкин)
```

А если нас интересуют все Васи, кроме Пупкина, это будет:

```
Вася (?!Пупкин)
```

Если же мы китайцы и пишем сперва фамилию, а потом имя, и при этом нас тоже интересует Вася, то:

```
(?<=Пупкин) Вася          - найдет всех Пупкиных Василиев
(?<!Пупкин) Вася - найдет всех Василиев, кроме Пупкиных
```