

# Регулярные выражения

История. Flavors. Синтаксис



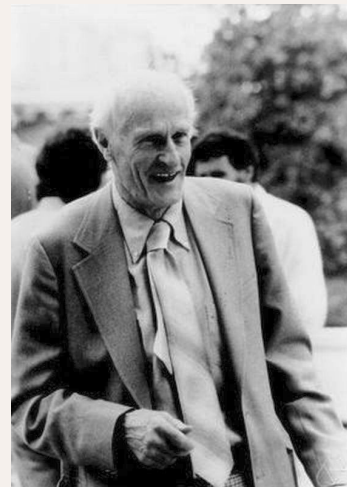


01

# Базовые концепты

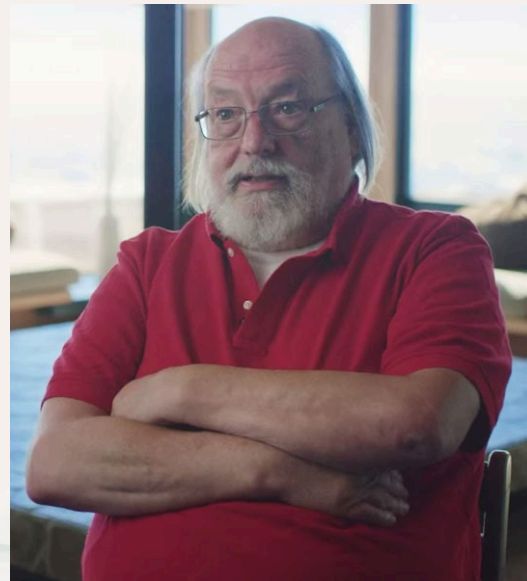
# Немного истории

- В 1950-х гг. XX века американский математик Стивен Коул Клини (Stephen Cole Kleene) формализовал концепт регулярного языка.
- Вспоминаем: это понятие из теории формальных языков Хомского.
- К какому типу формальных языков относится регулярный язык?
- Какие автоматы умеют распознавать регулярные языки?



# Немного истории

- К концу 70-х регулярные выражения обрели популярность
- Одним из первых применений был поиск по текстовому файлу в редакторе
- Кен Томпсон реализовал текстовый поиск с помощью регулярных выражений в коде и создал `grep`
- `g/re/p` = Global search for Regular Expression and Print matching lines



# Есть две задачи



## Поиск по тексту

♥ grep ♥

JULIA EVANS  
@b0rk

grep lets you search files for text

```
$ grep bananas foo.txt
```

Here are some of my favourite grep command line arguments !

**-i** case insensitive

**-A** Show context for your search.  
**-B** `$ grep -A 3 foo`  
**-C** will show 3 lines of context after a match

**-E** use if you want regexps like `".+"` to work. otherwise you need to use `".\+"`  
aka egrep

**-v** invert match: find all lines that don't match

**-l** only show the filenames of the files that matched

**-F** don't treat the match string as a regex  
eg `$ grep -F ...`

**-r** recursive! Search all the files in a directory.

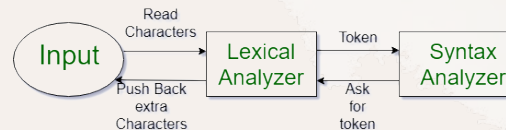
**-o** only print the matching part of the line (not the whole line)

**-a** search binaries: treat binary data like it's text instead of ignoring it!

grep alternatives  
`ack` `ag` `ripgrep`  
(better for searching code!)



## Компиляция программного кода





# Идея регулярных выражений

- Нужно искать в тексте последовательности
- Например, телефонные номера
- Или все словоформы какой-нибудь парадигмы
- Или варианты написания одного слова
- Или все слова (в противовес знакам пунктуации и символам)

Будем описывать то, что нам нужно, с помощью специальных символов



# Идея регулярных выражений

Например, хотим искать телефонные номера. Допустим, наши номера могут выглядеть так:

- 8(905)1234567
- +7(905)1234567
- 89051234567
- 8 905 1234567
- ...






%d %d %d %d %d %d %d %d %d %d



+ %d %d %d %d %d %d %d %d %d %d

+ %d \_|( %d %d %d \_|) %d %d %d %d %d %d %d



+ %d \_|( %d %d %d \_|) %d %d %d \_|- %d %d \_|- %d %d

# Flavors

IEEE POSIX стандарт:

## Basic RE

Минимум  
возможностей

## Extended RE

Добавляет символы  
+, |, ?, не нужно  
экранировать скобки

PCRE:

## Perl

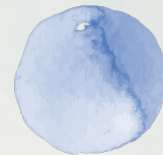
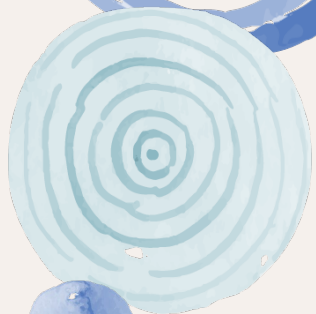
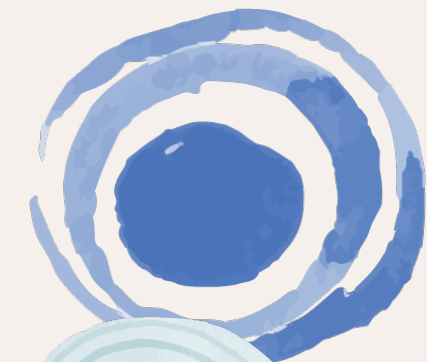
От разработчиков языка  
Perl. Добавили  
lookarounds

## PCRE

Реализации,  
совместимые с Perl-  
версией



# Синтаксис 02



# Что имеем

- Метасимволы
- Специальные последовательности
- Квантификаторы
- Классы символов
- Группы
- Проверки (lookarounds) – не является частью регулярного языка

# Метасимволы

- Обычно любой символ обозначает самого себя
- Но есть некоторые символы, которые означают что-нибудь еще. Вот они:
  - `\` - экранирование
  - `.` - любой символ (кроме переноса на новую строку) `т` `.` `ч` `к` `а` - точка, тучка, ...
  - `^` - начало строки, по которой ищем
  - `$` - конец строки, по которой ищем
  - `?`, `*`, `+`, `{}` - квантификаторы `а` `+` `й` - ай, аай, аааай, ...
  - `[]` - классы символов
  - `()` - группы
  - `|` - оператор «или» `а` `|` `б` - а, б



# Начало и конец строки

- Ищем в строке: «мама мыла раму, мама устала, бедная мама»

- мама мыла раму, мама устала, бедная мама

м а м а

- мама мыла раму, мама устала, бедная мама

^ м а м а

- мама мыла раму, мама устала, бедная мама

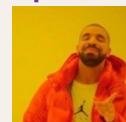
м а м а \$

# Спецпоследовательности

- **Один** любой символ какой-то определенной категории (в таблице Unicode):
- `\w` – все, что отнесено в категорию «буква» или «цифра»
- `\d` – все, что отнесено в категорию «цифра» (0123456789)
- `\s` – любые пробельные символы, включая перенос строки
- `\W` – все, кроме `\w`
- `\D` – все, кроме `\d`
- `\S` – все, кроме `\s`
- `\b` и `\B` – (не) граница между `\w` и `\W`.  
Не занимает своей позиции!

`\b` `р` `а` `м` `а` `\b`

«панора**м**а», «деревянная **р**ама»



# Квантификаторы

- Обычно любой символ в регулярном шаблоне обозначает только **один** элемент
- Квантификаторы это изменяют

- $\boxed{?}$  - 0 или 1 таких символов 

к	о	т	.	?
---	---	---	---	---

 = кот, кот<sup>а</sup>, кот<sup>у</sup>, кот<sup>ы</sup>

- $\boxed{*}$  - 0 или как можно больше 

к	о	т	.	*
---	---	---	---	---

 = кот, кот<sup>а</sup>, кот<sup>ов</sup>, кот<sup>ами</sup>

- $\boxed{+}$  - 1 или как можно больше 

к	о	т	.	+
---	---	---	---	---

 = кот<sup>а</sup>, кот<sup>ами</sup>, кот<sup>ики</sup>

- $\{x,y\}$  – от <sup>х</sup> до <sup>у</sup>

к	о	т	.	{1,2}
---	---	---	---	-------

 = кот<sup>ы</sup>, кот<sup>ов</sup>

- Может быть:  $\{,y\}$  = от 0 до <sup>у</sup> или  $\{x,\}$  = от <sup>х</sup> до бесконечности, или  $\{x\}$  – точно <sup>х</sup>

# Квантификаторы

- Обычно все квантификаторы стараются найти как можно больше символов
- Поэтому они называются **жадными**
- Можно изменить их поведение и сделать их **ленивыми**
- Нужно поставить справа ?

к о т . \*

найдет и кот, и котенька, но предпочтет последнее

к о т . \* ?

найдет и кот, и котенька, но предпочтет первое

# Классы символов

- Когда нам подходит в определенной позиции несколько вариантов, можем воспользоваться оператором «или»:

т	(о   у)	ч	к	а
---	---------	---	---	---

 = точка или тучка

- Но если вариантов слишком много, можно воспользоваться классом:

т	[аоу]	ч	к	а
---	-------	---	---	---

 = точка, тачка, тучка

- Класс символов занимает только **одно** место. Квантификатор относится к классу
- [аоу]+** = ааа, ооо, ууу, аоа, уао, аоу... (порядок и набор не имеет значения)
- Внутри класса метасимволы теряют свою силу (все, кроме `\`).
- Некоторые символы ее обретают или меняют:
  - `^` в начале класса (`[^аоу]`) означает «все, кроме...»
  - `-` в середине означает диапазон: `[а-я]` = все буквы от а до я (по юникоду!)

# Группы

- Действие оператора «или» обычно распространяется на весь шаблон:

`т о | у ч к а` = то или учка

`т (о | у) ч к а` = точка или тучка

- Группа (в круглых скобках) может его ограничить. Но это не единственное применение круглых скобок!
- Обычно регулярный язык позволяет извлекать содержимое группы отдельно:

`(. *) \. г у` = для строки «mail.ru» вернет полный результат `mail.ru` и содержимое группы `mail`

- Поэтому и говорят «группа с захватом содержимого» (capturing group)

# Группы

- Группы в шаблоне нумеруются (по открывающей скобке) начиная с 1. На группы можно ссылаться внутри шаблона. Например:

`(\w+)` и `\1` = «сделал и сделал», «шаблон и шаблон»

- Группе можно дать имя: `(?P<имя>\w+)`  
а обратно сослаться на нее как: `(?P=имя)`
- Группу можно также сделать группой без захвата содержимого (non-capturing):  
`(?:\w+)`
- Мы не сможем получить доступ к содержимому такой группы и не сможем сослаться на нее.
- Это бывает иногда нужно из-за особенностей реализации регулярок в питоне



# Проверки

- Проверка ставит нашей регулярке какое-то условие, при этом само условие в результат не попадает.
- Проверки бывают:
  - Positive lookahead       $\text{Вася}(?= \text{Пупкин}) = \text{Вася Пупкин} \checkmark, \text{Вася Сидоров} \times$
  - Negative lookahead       $\text{Вася}(?! \text{Пупкин}) = \text{Вася Пупкин} \times, \text{Вася Сидоров} \checkmark$
  - Positive lookbehind       $(?<= \text{Пупкин}) \text{Вася} = \text{Пупкин Вася} \checkmark, \text{Сидоров Вася} \times$
  - Negative lookbehind       $(?<! \text{Пупкин}) \text{Вася} = \text{Пупкин Вася} \times, \text{Сидоров Вася} \checkmark$
- Бывает важно, когда мы хотим дать тексту в условии шанс попасть в следующий match: ведь регулярки работают линейно и для строки «aaaa» регулярка aa найдет только два вхождения.

# Полезные ссылки

- <https://regex101.com/>
- <https://www.regular-expressions.info/reference.html>
- <https://regexcrossword.com/>