BI296: Linux and Shell Programming

# Lecture 03: Regular Expression

Maoying,Wu

ricket.woo@gmail.com

Dept. of Bioinformatics & Biostatistics
Shanghai Jiao Tong University

Spring, 2018

# Lecture Outline

- Regular Expression (正则表达式)
  - Notations （概念）
  - Types of REGEX
  - Metacharacters (元字符)
  - Group Capturing (组捕获) and Backreference （后向引用）
  - Non-capturing groups (非捕获组) and zero-length assertions (零宽断言)
  - Cases (案例分析)

- Applications of REGEX （正则表达式应用）
  - `grep`: text matching
  - `sed`: streaming editor
  - `awk`: mini programming environment

# Regular expression: Notations

**Regular expression (正则表达式)**
Also called patter matching (模式匹配), used for matching, searching, and replacing the given text pattern in a given set of strings.

**String Pattern (字符串模式)**
A string which can represent a set of possible strings.

**Metacharacter (元字符)**
Some special characters used for reprenting some characters.

**Greedy/Lazy matching (贪婪/惰性匹配)**
Finding the maximum/minimum matching (最大/最小匹配方式).

## Examples

- `ps -aux | grep mysql`
- `sed -i ’s/ˆ$//g’ filename`
- `awk ’/ˆATOM/{print $2}’ 1xhu.pdb`

# History of REGEX

- 1943: Warren McCulloch and Walter Pitts - Nervous system models (i.e., how a machine could be built like a brain)
- 1956: Stephen Kleene describes these models with an algebra called ”regular sets” and creates a notation to express them called ”regular expressions”
- 1968: Ken Thompson implements regular expressions in `ed`:
  - `g/REGEX/p`: g - globally, p - print
  - Global Regular Expression Print: `grep`
  - Became widely used in `awk, vim, emacs`, etc.
- 1986: POSIX (Portable Operating System Interface) - standard
  - Basic Regular Expressions (BREs)
  - Extended Regular Expressions (EREs)
- 1986: Henry Spencer releases a `regex` library written in C.
- 1987: Larry Wall released Perl
  - Used `regex` library, and added more powerful features
  - Perl-Compatible Regular Expression (PCRE)

# Conventions and Modes

## Conventions (传统表示方法)

- `grep`: 'regex' (enclosed in single quotes)
- `sed`: /regex/ (encloded in forward slashes)
- `awk`: /regex/ (enclosed in forward slashes)

## Modes (工作模式)

- **REGULAR** mode (一般模式): 'regex', /regex/
- **MULTILINE** mode (多行模式): '(?m)regex'(`grep -Pz`), /regex/m (`sed -z`)
- **DOT_AS_ALL** mode (点全匹配模式): '(?s)regex' (`grep -Pz`), /regex/s
- **CASE_INSENSITIVE** mode (大小写不敏感模式): '(?i)regex' (`grep -P`), /regex/i
- **GLOBAL** mode (全局模式): /regex/g (`sed`), /regex/g

# Literal Characters: Plain text

- Strings
  - `/gene/` matches ”gene”;
  - `/gene/` also matches the first four letters of ”generation”;
  - Similar to searching in a word processor
- Case-sensitive (by default)
  - `gene` does not match ”Generation”;
- Non-global matching will prefer the leftmost match.
  - `/cat/` matches ”The cow, camel and cat communicate with each other.”

# Position Anchors (定位元字符

| Metachar | Description | Examples |
|---|---|---|
| `^` | matching the start of a line. | `^ATOM` |
| `$` | matching the end of a line. | `\.$` |
| `\<` | matching the start of a word. | `\<root` |
| `\>` | matching the end of a word. | `root\>` |
| `\b` | matching the boundary of a word. | `\broot\b` |

## Note

- When located not at the starting of the regex, `^` has no special meaning.
- Similarly, when not located at the end of the regex, `$` has no special meaning.
- Here boundary-of-word means the non-alphanumeric characters.

# Metacharacters (元字符): Characters with special meanings

| Metachar | Description | Examples |
|---|---|---|
| . | Any single character except the newline. | atg.ccc. |
| .? | **0-1 repeats** of the preceding char. | te?a |
| .* | **0+ repeats** of the preceding char. | te*a |
| .+ | **1+ repeats** of the preceding char. | te*t |
| [...] | **Positive set**, matching one. | t[aeiou]n |
| [^...] | **Negative set**, matching one. | t[^ae]n |
| (...) | **Group** the characters. | atg([actg][actg][actg])+tca |
| \1,\2,... | **Backreference**. | atg(att)\1acc |
| (..\|..) | **Alternation**. | (abc\|xyz) |
| {m[,[n]]} | Specifying the number of repeats. | atg([actg]{3}){5,10}tca |

## Note

- ., ?, *, + keeps their literal meaning when located within a set [.?*+]
- Sometimes – has special meaning, like [3-8] and [a-f].
- However – in [-abcf-] regains its literal meaning.
- The ^ sign in [a^bc] has no special meaning.

# Repetition Metacharacters

## Examples

- `/apples?/` matches ”apple” and ”apples”, but not ”applesssss”
- `/apples+/` matches ”apples” and ”applesssss”, but not ”apple”
- `/apples*/` matches ”apple”, ”apples” and also ”applesssss”
- `\d\d\d\d?` matches numbers with 3-4 digits.
- `\d\d\d\d*` matches numbers with 3 or more digits.
- `\d\d\d\d+` matches numbers with 4 or more digits.
- `colou?r` matches either ”color” or ”colour”.
- `\d{4,8}` matches numbers with 4-8 digits.
- `\d{4}` matches numbers with exactly 4 digits.
- `\d{4,}` matches numbers with at least 4 digits.
- `0\d{2,3}-\d{6,8}` matches most Chinese phone numbers.

## Support

- `*` is supported in all regex engines.
- `?` and `+` are not supported in BREs.

# Shorthand Character Sets

| Shorthand | Meaning | Equivalent |
|---|---|---|
| `\d` | Digit | `[0-9]` |
| `\w` | Word character | `[a-zA-Z0-9_]` |
| `\s` | Whitespace | `[ \t\r\n]` |
| `\D` | Not digit | `[^0-9]` |
| `\W` | Not word character | `[^a-zA-Z0-9_]` |
| `\S` | Not whitespace | `[^\t\r\n ]` |

# Posix Bracket Expressions

| Class | Meaning | Equivalent |
|---|---|---|
| `[:alpha:]` | Alphabetic characters | `A-Za-z` |
| `[:digit:]` | Numeric characters | `0-9` |
| `[:alnum:]` | Alphanumeric characters | `A-Za-z0-9` |
| `[:lower:]` | Lowercase alphabetic characters | `a-z` |
| `[:upper:]` | Uppercase alphabetic characters | `A-Z` |
| `[:punct:]` | Punctuation characters | |
| `[:space:]` | Space characters | `\s` |
| `[:blank:]` | Blank characters (space,tab) | |
| `[:print:]` | Printable characters,space | |
| `[:graph:]` | Printable characters,no space | |
| `[:cntrl:]` | Control characters (non-printable) | |
| `[:xdigit:]` | Hexadecimal characters | `A-Fa-f0-9` |

- Correct: `[[:alpha:]]` or `[^[:alpha:]]`
- Incorrect: `[:alpha:]`

# Three Versions of REGEX Syntax

- Basic Regular Expression (BRE，基本正则表达式)
- Extended Regular Expression (ERE，扩展正则表达式)
- Perl-Compatible Regular Expression (PCRE，Perl正则表达式)

## BRE vs. ERE vs. PCRE

- In BRE the meta-characters `?`, `+`, `{`, `|`, `(`, and `)` give their literal meanings.
- Instead BRE use the backslashed versions `\?`, `\+`, `\{`, `\|`, `\(`, and `\)` to represent the special meanings.
- ERE supports all of the above metacharacters.
- PCRE supports lazy matching（惰性匹配）, zero-length assertion（零宽断言）and named capturing（命名组捕获）.
- `grep` uses BRE by default; `grep` need to specify the “-E” option to enable ERE; `grep` need to specify the “-P” option to enable PCRE.
- Both `sed` and `awk` do not support PCRE.

# BRE: Examples

```
# containing, not containing
grep -e "root" passwd
grep -v -e "root" passwd

# start/end with
grep -e "^root" passwd
grep -e "nologin$" passwd

# either... or...
grep -e "root\|bio" passwd
grep -e "root" -e "bio" passwd

# repeats, group, backreference
grep -e "[0-9]\{8\}" passwd
grep -e "\(root\).*\1" passwd
grep -e "\(root\|bio\).*\1" passwd
grep -e "\(o\{2,\}\).*\1" passwd
grep -e "[^0-9]\([0-9]\{2\}\)\([^0-9]\)\1\2" passwd

# escape characters
grep -e "\." passwd
grep -e "[*(0-9[]" passwd
grep -e "^\(root\).*" passwd
grep -e "\([aeiou]\)\{2,\}" passwd
```

# ERE: Examples

```
# alternation
grep −E ’root|bio’ passwd

# repeats {}
grep −E ’[0−9]{8}’ passwd

# group (), +
grep −E ’(root).+\1’ passwd
grep −E ’(root|bio).+\1’ passwd
grep −E ’(o{2,}).+\1’ passwd


grep −E ’[ˆ0−9]([0−9]{2})([ˆ0−9])\1\2’ passwd
grep −E ’o+’ passwd}
```

# Capturing Groups (捕获组)

*The stuffs captured by regex enclosed by parentheses.*

| Expressions | Description |
|---|---|
| `(exp)` | **Non-named capturing group (非命名捕获组)** matching `exp` |
| `(?<name>exp)` | **Named-capturing group (命名捕获组)** with name `name` |
| `(?'name'exp)` | **Named-capturing group** with name `name` matching `exp` |
| `(?:exp)` | **Non-capturing group (非捕获组)** matching `exp` |
| `\1,\2,...` | **Backreference (后向引用)** of the non-named capturing groups |
| `\k<name>` | **Backreference (后向引用)** of the named capturing group |
| `\k'name'` | **Backreference (后向引用)** of the named capturing group |

## Examples

- `grep -P "^(root).*(?=\1)" /etc/passwd`
- `grep -P "^(?<name>root).*(?=\k<name>)" /etc/passwd`
- `grep -P "^(?'name'root).*(?=\k'name')" /etc/passwd`

# Zero-Length Assertion (零宽断言)

*a.k.a.* ***LOOK-AROUND****, ONLY match the position, but NOT a real string.*

| **Assertions** | **Description** |
|---|---|
| `(?=exp)` | **positive look-ahead (正向先行断言)**, matching the position before `exp` |
| `(?!exp)` | **negative look-ahead (负向先行断言)**, matching the position not before `exp` |
| `(?<=exp)` | **positive look-behind (正向后行断言)**, matching the position after `exp` |
| `(?<!exp)` | **negative look-behind (负向后行断言)**, matching the position not after `exp` |

## PCRE Examples

Note: The `exp` in look-behind assertion should have fixed length.

- `echo "adhd" | grep -P "(?<=h)d"`
- `grep -P "(?<=/)root" /etc/passwd`
- `grep -P "(?<!.)root" /etc/passwd`
- `grep -P "root(?=:)" /etc/passwd`
- `grep -P "root(?!:)" /etc/passwd`

# Regular Expression: Examples

`/^[0-9]+$/`:
matches any input line that consists of only digits.

`/^[0-9][0-9][0-9]$/`
exact three digits

`/^(\+|-)?[0-9]+\.?[0-9]*$/`
a decimal number with an optional sign and optional fraction

`/^[+-]?[0-9]+[.]?[0-9]*$/`
also a decimal number with an optional sign and optional fraction

`/^[+-]?([0-9]+[.]?[0-9]*|[.][0-9]+)([eE])?$/`
a floating point number with optional sign and optional exponent

`/^[A-Za-z]|[A-Za-z0-9]*/`
a letter followed by any letters or digits

`/^[A-Za-z]$|^[A-Za-z0-9]*$/`
a single letter or any length of alphanumeric characters

`/^[A-Za-z][0-9]?$/`
a letter followed by 0-1 digit

# Next we will talk about ...

1. Regular Expression
2. Regular expression: Introduction
3. Regular expression: Applications
   - grep
   - sed
   - awk

# Using grep to find patterns in a text

## Synopsis (用法)

- `grep -oeEP 'PATTERN' FILENAME`
- `SOME_COMMAND | grep -oeEP 'PATTERN'`

## PATTERN (模式)

1. **PATTERN** can be any regular string
2. **PATTERN** can include escape character
3. **PATTERN** can include some metacharacters with special meanings.
4. **PATTERN** should be enclosed in single quotes.

## Options (常用选项)

- **-e**: use BRE
- **-E**: use ERE
- **-P**: use PCRE

# grep: A multiline matching example

```
grep -Pzo '(?s)^(\s*)\N*main.*?\{.*?^\1\}' test.c
```

| keywords | Description |
|---|---|
| `-P` | activate PCRE for `grep`. |
| `-z` | activate multiline mode. |
| `-o` | print only matching. |
| `(?s)` | activate PCRE_DOTALL. |
| `\N` | match anything except newline. |
| `.*?` | suppress greedy matching mode. |
| `^` | match start of line. |

# Greedy vs. Non-greedy Match (贪婪匹配vs 非贪婪匹配)

## Examples

- `echo "page 2567" | grep -Po ".*(?!(\w+))"`
- `echo "page 2567" | grep -Po ".*?(?!(\w+?))"`
- `echo "page 2567" | grep -Po ".*(?=(\d+))"`
- `echo "page 2567" | grep -Po ".*?(?=(\d+))"`

- Non-greedy mode is only supported in PCRE.
- Standard repetition quantifiers are greedy - expression tries to match the longest possible string.
- Defers to achieving overall match.
  - `/.+\.jpg/` matches ”filename.jpg”
  - The `+` is greedy, but ”gives back” the ”.jpg” to make the match.
  - Think of it as rewinding or backtracking.

# What would this match?

```
echo "Page␣2687" | grep -P '.*?[0-9]*?'
echo "Page␣2687" | grep -P '.+?[0-9]*?'
```

# Next we will talk about ...

1. Regular Expression
2. Regular expression: Introduction
3. Regular expression: Applications
   - grep
   - sed
   - awk

# sed: Stream Editor

## Synopsis

```
sed [-e script] [-f scriptfile] [-n] [files...]
```

| | |
|---|---|
| **-e** | Followed by inline scripts, default BRE |
| **-n** | Suppress automatic printing of pattern space until the **p** action. |
| **-f** | Read scripts from a sed file. |
| **-i** | Edit files in place. |
| **-r** | Using extended regular expression. |
| **files** | The files for analyzing, ‘-’ for **stdin**. |

## invoking sed

```
sed -e ’[address1[,address2]][action]’ infiles
sed -e ’command1;command2’ infile # output results to screen
sed -e ’command1;command2’ infiles > outfile # save results
command | sed -e ’command-sets’ | command # piping
sed -f sedfile infile > outfile # command saved in a file
```

# Addresses

| Address type | Meaning |
|---|---|
| number | Match only the specified line number. |
| $ | Match the last line. |
| first~step | Match every step lines starting from first. |
| /regexp/ | Match lines matching the regular expression regexp. |
| \cregexpc | Match lines matching the regular expression regexp. |
| 0,addr2 | read until the first match of addr2 (can be number or regexp). |
| addr1,+N | Match addr1 and the following N lines. |
| addr1,~N | Match addr1 and continue until the line number is a multiple of N. |

### Example

```
sed -n -e ’1,˜5p’ /etc/passwd
sed -n -e ’1˜5p’ /etc/passwd
sed -n -e ’1,+5p’ /etc/passwd
sed -n -e ’1,/root/p’ /etc/passwd
sed -n -e ’0,/root/p’ /etc/passwd
```

# Two Data Buffers (数据缓存空间)

- **Pattern Space (模式空间)**: By default the streaming data will be stored into the pattern space line by line. And the data will be output to screen.
- **Hold Space (保留空间)]**: The buffer for storing the temporary data.

**Workflow (sed的一般工作流程)**

(1) Stores the current line in the pattern space;
(2) Deals with contents in the pattern space according to specified actions;
(3) Print out the contents in pattern space;
(4) Clear the contents in the pattern space;
(5) Start next cycle.

# sed actions

| Action | Description |
|---|---|
| d | Delete pattern space and start next cycle. |
| h/H | Copy/append pattern space to hold space. |
| g/G | Copy/append hold space to pattern space. |
| x | Exchange the contents of the hold and pattern spaces. |
| p | Print the contents in pattern space. |
| P | Print the contents in pattern space up to the first newline. |
| q | Quit the current cycle. |
| s/RE/string/ | Replacement. |
| y/chars/chars/ | Translate. |
| c | Change the pattern space with something. |
| i | Insert something before the pattern space. |
| a | Append something into the pattern space. |

## Examples

```
sed -n ’1{h;n;x;H;x};p’ filename # exchange line 1 and 2
sed -n -e ’1!G;h;$p’ filename # ==tac
sed -e ’1!G;h;$!d’ filename # ==tac
```

# Branch Commands

- `:label`:
  Set label for `b` and `t/T` commands.
- `b label`:
  Branch to label; if label is omitted, branch to end of script.
- `t label`:
  If a `s///` has done a successful substitution since the last input line was read and since the last `t` or `T` command, then branch to label; if label is omitted, branch to end of script.
- `T label`:
  If no `s///` has done a successful substitution since the last input line was read and since the last `t` or `T` command, then branch to label; if label is omitted, branch to end of script.

# sed: Converting fastq to fasta

## FASTQ file

```
@SRR018006.2016 GA2:6:1:20:650 length=36
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNGN
+SRR018006.2016 GA2:6:1:20:650 length=36
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!+!
@SRR018006.19405469 GA2:6:100:1793:611 length=36
ACCCGCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
+SRR018006.19405469 GA2:6:100:1793:611 length=36
7);;).;);;/;*.2>/@@7;@77<..;)58)5/>/
```

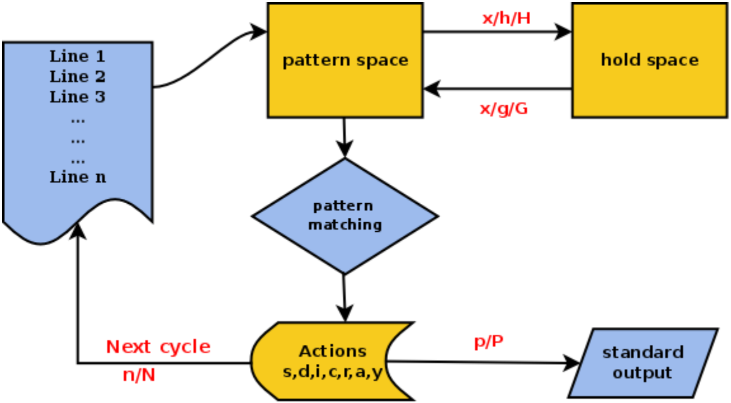
## solution

```
sed ’/@/!d;s//>/;N’ test.fastq > test.fasta
```

# sed: Another solutions

```
#!/bin/sed -r -f
# Read a total of four lines into buffer
$b error
# if empty, jump to :error
N;$b error
# next line, if empty, jump to :error
N;$b error
# ...
N
# next line
# Parse the lines
/@(([ ]*).*)(\n[ACGTN]*)\n\+\1\n.*$/{
# Output id and sequence for FASTA format.
s//>\2\3/
b
}
:error
i\
Error parsing input:
q
```

# sed: Summary


Line 1
Line 2
Line 3
...
...
...
Line n
pattern space
x/h/H
hold space
x/g/G
pattern matching
Actions s,d,i,c,r,a,y
p/P
standard output
Next cycle
n/N

# Next we will talk about ...

1. Regular Expression
2. Regular expression: Introduction
3. Regular expression: Applications
    - grep
    - sed
    - awk

# awk: An interpreter language

- Named from three authors: Alfred Aho, Peter Weinberger, and Brian Kernighan.
- Using C-style syntax
- Support regular expression (正则表达式) and associative arrays (关联数组)
- Good at editing field data

### Example

```
ls -l | awk ’{print $5,$8}’
ls -l | awk ’{print "File",$8,"size =",$5, "Bytes."}’
```

# awk: Built-in variables

| built-in variable | Description |
|---|---|
| **$0** | The whole line. |
| **$1** | The first field of current line. |
| **$n** | The *n*-th field of current line. |
| **ARGC** | Input arguments count. |
| **ARGV** | Input argument vector. |
| **FILENAME** | Name of current input file. |
| **NR** | Records number up to now. |
| **FNR** | Record number of current file. |
| **NF** | Number of fields for current record. |
| **FS/IFS** | Input field separator. |
| **OFS** | Output field separator. |
| **OFMT** | Output format for numbers, default %.6g. |
| **RS** | Input record separator, default newline. |
| **ORS** | Output record separator, default newline. |
| **RSTART** | Index of first character matched by match. |
| **RLENGTH** | Match length of string match by match. |
| **SUBSEP** | Subscript separator, default `\034`. |

# Three kinds of blocks: BEGIN{},{},END{}

```
BEGIN
{
    actions
}
[PATTERN]
{
    actions
}
END
{
    actions
}
```

- **BEGIN** will be executed prior to the manipulation of the target file.
- **MAIN** block will be executed on the file line by line.
- **END** will be executed after the file reach the end.

# Example: Setting FS

```
#!/usr/bin/awk -f
# file: test.awk
BEGIN
{
    FS="[:-]"
}
{
    for (i=1; i<=NF; i++) print $i;
}
END
{
    print "The", FILENAME, "has", NR, "rows."
}
```

### Run the script file

```
echo -e "ab-cd:ef\ngh:ij-kl" | awk -f test.awk
```

# Another awk Example

```
#!/bin/awk -f
# test2.awk
BEGIN
{
    FS=":"
}
{
    if ($2 == "")
    {
        print $1 ":␣no␣password";
        total++;
    }
}
END
{
    print "Total␣no-password␣account␣=␣", total
}
```

## Run

```
chmod u+x test2.awk
./test2.awk /etc/shadow
```

# awk patterns: relational operators

| **Regexs** | Meaning |
|---|---|
| `$1~/regex/{actions}` | if the field 1 matches regex, |
| `$1!~/regex/{actions}` | if the field 1 does not match, |
| `/regex/{actions}` | if the whole line matches |
| `!/regex/{actions}` | unless the whole line matches |

| **Operators** | Meaning |
|---|---|
| `$1==5{actions}` | Equal |
| `$1!=5{actions}` | Not equal |
| `$1>5{actions}` | Greater than |
| `$1>=5{actions}` | Greater than or equal to |
| `$1<5{actions}` | Less than |
| `$1<=5{actions}` | Less than or equal to |
| `$1<5 && $2>6{actions}` | Conditional AND |
| `$1<5 \|\| $2>6{actions}` | Conditional OR |

# Control Flow Statements

| **command and short description** |
|---|
| `{statements}`: Execute all the statements in the brackets. |
| `if(expression)statement`: If expression is true, execute. |
| `if(expression)statement1 else statement2`: if-condition. |
| `for(expression1;expression2;expression3)statement`: C-style for. |
| `for(variable in array)statement`: in-style for. |
| `while(expression)statement`: while-loop. |
| `do statement while(expression)` do-while-loop. |
| `break`: immediately leave innermost. |
| `continue`: start next iteration of innermost. |
| `next`: start next iteration of main input loop. |
| `exit`: exit |
| `exit expression`: go immediately to END. |

# Associative arrays (关联数组)

- All awk arrays are in fact associative arrays (关联数组).
- The subscript (or the index) can be either numeric or string, but they are actually strings.
-

```
#!/bin/awk -f
BEGIN
{
    for (i=0; i<10; i++)
    {
        for (j=0; j<10; j++)
        {
            prod[i][j] = i * j;
        }
    }
    for (i=0; i<10; i++)
    {
        for (j=0; j<=i; j++)
        {
            printf("%dx%d=%2d␣", i, j, prod[i][j]);
        }
        print;
    }
}
```

# Builtin Arithmetic Functions

| **Functions** | **Description** |
|---|---|
| `atan2(y,x)` | arctangent of $y/x$ in the range $-\pi$ to $\pi$ |
| `cos(x)` | cosine of $x$, with $x$ in radians. |
| `exp(x)` | exponential function of $x$, $e^x$ |
| `int(x)` | integer part of $x$; truncated towards 0 |
| `log(x)` | natural logarithm of $x$ |
| `rand()` | random number $0 \le r \le 1$ |
| `sin(x)` | sine of $x$, with $x$ in radians |
| `sqrt(x)` | square root of $x$ |
| `srand(x)` | $x$ is new seed for `rand()` |

# Built-in string functions

| **Functions** | **Description** |
|---|---|
| `gsub(r,s)` | Substitute `s` for `r` globally in `$0`. |
| `gsub(r,s,t)` | Substitute `s` for `r` globally in string `t`. |
| `index(s,t)` | First position of string `t` in `s`, 0 otherwise. |
| `length(s)` | Length of string `s`. |
| `match(s,r)` | Substring match. sets RSTART and RLENGTH. |
| `split(s,a)` | split `s` into array `a` using FS; return length(a). |
| `split(s,a,fs)` | split `s` into array `a` using `fs`. |
| `sprintf(fmt,exprs)` | return string according to format `fmt`. |
| `sub(r, s)` | substitute `s` by `r`. |
| `sub(r,s,t)` | substitute `s` by `r` in `t`. |
| `substr(s,p)` | return suffix of `s` starting at `p`. |
| `substr(s,p,n)` | return substring of `s` starting from `p` with length `n`. |

# A short awk script without input files

```
#!/bin/awk -f
# seq.awk - print sequences of integers
# input: arguments q, p q, or p q r; q >= p & r > 0
# output: integer 1 to q, in step of r
BEGIN
{
    if (ARGC == 2)
    for (i = 1; i <= ARGV[1]; i++) print i
    else if (ARGC == 3)
    for (i=ARGV[1]; i <= ARGV[2]; i++) print i
    else if (ARGC == 4)
    for (i=ARGV[1]; i <= ARGV[2]; i += ARGV[3]) print i
}
```

## Run

```
awk -f seq.awk 10
awk -f seq.awk 1 10
awk -f seq.awk 1 10 1
```

# Compute column sums

```
# sum1.awk - print column sums
# input: rows of numbers
# output: sum of each column
#
{
    for ( i = 1; i <= NF; i++) sum[i] += $i
    if (NF > maxfld) maxfld = NF
}
END
{
    for (i=1; i <= maxfld; i++)
    {
        printf("%g", sum[i])
        if (i < maxfld) printf("\t")
        else printf("\n")
    }
}
```

# Draw a histogram

```
#!/bin/awk -f
# histogram.awk
# input: numbers between 0 and 100
{
      x[int($1/10)]++
}
END
{
      for (i=0; i < 10; i++)
      printf("␣%2d␣-␣%2d:␣%3d␣%s\n", 10*i, 10*i+9, x[i], rep(x[i], "*"))
}
function rep(n, s, t)
{
      while (n-- > 0) t = t s
      return t
}
```

## Run scripts

```
chmod u+x histogram.awk
awk ’
BEGIN {
for (i=1; i<=200; i++) print int(100*rand())
}’ | ./histogram.awk
```