Go by Example: Regular Expressions

Go offers built-in support for regular expressions. Here are some examples of common regexp-related tasks in Go.

```
package main

import (
    "bytes"
    "fmt"
    "regexp"
)

func main() {

    match, _ := regexp.MatchString("p([a-z]+)ch", "peach")
    fmt.Println(match)

    r, _ := regexp.Compile("p([a-z]+)ch")

    fmt.Println(r.MatchString("peach"))

    fmt.Println(r.FindString("peach punch"))

    fmt.Println("idx:", r.FindStringIndex("peach punch"))

    fmt.Println(r.FindStringSubmatch("peach punch"))

    fmt.Println(r.FindStringSubmatchIndex("peach punch"))

    fmt.Println(r.FindAllString("peach punch pinch", -1))

    fmt.Println("all:", r.FindAllStringSubmatchIndex(
        "peach punch pinch", -1))

    fmt.Println(r.FindAllString("peach punch pinch", 2))

    fmt.Println(r.Match([]byte("peach")))

    r = regexp.MustCompile("p([a-z]+)ch")
    fmt.Println("regexp:", r)

    fmt.Println(r.ReplaceAllString("a peach", "<fruit>"))

    in := []byte("a peach")
    out := r.ReplaceAllFunc(in, bytes.ToUpper)
    fmt.Println(string(out))
}
```

This tests whether a pattern matches a string.

Above we used a string pattern directly, but for other regexp tasks you'll need to `Compile` an optimized `Regexp` struct.

Many methods are available on these structs. Here's a match test like we saw earlier.

This finds the match for the regexp.

This also finds the first match but returns the start and end indexes for the match instead of the matching text.

The `Submatch` variants include information about both the whole-pattern matches and the submatches within those matches. For example this will return information for both `p([a-z]+)ch` and `([a-z]+)`.

Similarly this will return information about the indexes of matches and submatches.

The `All` variants of these functions apply to all matches in the input, not just the first. For example to find all matches for a regexp.

These `All` variants are available for the other functions we saw above as well.

Providing a non-negative integer as the second argument to these functions will limit the number of matches.

Our examples above had string arguments and used names like `MatchString`. We can also provide `[]byte` arguments and drop `String` from the function name.

When creating global variables with regular expressions you can use the `MustCompile` variation of `Compile`. `MustCompile` panics instead of returning an error, which makes it safer to use for global variables.

The `regexp` package can also be used to replace subsets of strings with other values.

The `Func` variant allows you to transform matched text with a given function.

```
$ go run regular-expressions.go
true
true
peach
idx: [0 5]
[peach ea]
```

```
[0 5 1 3]
[peach punch pinch]
all: [[0 5 1 3] [6 11 7 9] [12 17 13 15]]
[peach punch]
true
regexp: p([a-z]+)ch
a <fruit>
a PEACH
```

For a complete reference on Go regular expressions check the `regexp` package docs.

Next example: JSON.

by Mark McGranaghan and Eli Bendersky | source | license