Strings and strconv Package

In this lesson, you'll study strings, strconv package, and the functions supported by them.

WE'LL COVER THE FOLLOWING

- Prefixes and suffixes
- Testing whether a string contains a substring
- Indicating the index a substring or character in a string
- Replacing substring
- Counting occurrences of a substring
- Repeating a string
- Changing the case of a string
- Trimming a string
- Splitting a string
    - On whitespaces
    - On a separator
- Joining over a slice
- Reading from a string
- Conversion to and from a string

Strings are a basic data structure, and every language has a number of predefined functions for manipulating strings. In Go, these are gathered in a package, `strings`. We'll discuss below some very useful functions one by one.

Prefixes and suffixes

`HasPrefix` tests whether the string `s` begins with a *prefix* `prefix`:

```
strings.HasPrefix(s, prefix string) bool
```

`HasSuffix` tests whether the string `s` ends with a *suffix* `suffix`:

HasSuffix tests whether the string s ends with a *suffix* suffix:

```
strings.HasSuffix(s, suffix string) bool
```

The following program implements these functions:

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    var str string = "This is an example of a string"
    fmt.Printf("T/F? \nDoes the string \"%s\" have prefix %s? ", str, "Th")
    fmt.Printf("\n%t\n\n", strings.HasPrefix(str, "Th"))    // Finding prefix

    fmt.Printf("Does the string \"%s\" have suffix %s? ", str, "ting")
    fmt.Printf("\n%t\n\n", strings.HasSuffix(str, "ting"))  // Finding suffix
}
```

Prefixes and Suffixes in a String

As you can see in the above code, we declare a string str and initialize it with **This is an example of a string** at **line 9**. At **line 11**, we used the function HasPrefix to find prefix **Th** in string str. The function returns *true* because str does start with **Th**. Similarly, at **line 14**, we used the function HasSuffix to find the suffix **ting** in string str. The function returns *false* because str does not end with **ting**. This also illustrates the use of the escape character \ to output a literal " with \" , and the use of 2 substitutions in a format-string.

Testing whether a string contains a substring

The function Contains returns *true* if substr is within s:

```
strings.Contains(s, substr string) bool
```

Indicating the index a substring or character in a string

Index returns the index of the *first* instance of str in s, or **-1** if str is not present in `s`:

```
strings.Index(s, str string) int
```

`LastIndex` returns the index of the *last* instance of `str` in `s`, or **-1** if `str` is not present in `s`:

```
strings.LastIndex(s, str string) int
```

If `ch` is a *non-ASCII* character, use:

```
strings.IndexRune(s string, ch int) int.
```

The following program finds the index of the substrings in a string:

```
package main
import (
    "fmt"
    "strings"
)

func main() {
    var str string = "Hi, I'm Marc, Hi."
    fmt.Printf("The position of the first instance of\"Marc\" is: ")
    fmt.Printf("%d\n", strings.Index(str, "Marc"))      // Finding first occurence
    fmt.Printf("The position of the first instance of \"Hi\" is: ")
    fmt.Printf("%d\n", strings.Index(str, "Hi"))        // Finding first occurence
    fmt.Printf("The position of the last instance of \"Hi\" is: ")
    fmt.Printf("%d\n", strings.LastIndex(str, "Hi"))    // Finding last occurence
    fmt.Printf("The position of the first instance of\"Burger\" is: ")
    fmt.Printf("%d\n", strings.Index(str, "Burger"))    // Finding first occurence
}
```

Position of Substring

As you can see in the above code, we declare a string `str` and initialize it with **Hi, I'm Marc, Hi.** at **line 8**. At **line 10**, we find the index of the first occurrence of **Marc** in `str` using the `Index` function that gives **8** as a result. Similarly, at **line 12**, we find the index of the first occurrence of **Hi** in `str` using the `Index` function, which is **0**, and its last occurrence by using `LastIndex` at **line 14**, which is **14**. At **line 16**, we find the first occurrence of **Burger** in `str` which gives **-1** because it doesn't exist.

Replacing substring

We can replace an old string with a new string like:

```
strings.Replace(str, old, new string, n int)
```

We can replace the first `n` occurrences of `old` in `str` by `new`. A copy of `str` is returned, and if `n` is **-1**, all occurrences are replaced.

Counting occurrences of a substring

`Count` counts the number of non-overlapping instances of substring `str` in `s` with:

```
strings.Count(s, str string) int
```

Run the below program as an example.

```
package main
import (
    "fmt"
    "strings"
)

func main() {
    var str string = "Hello, how is it going, Hugo?"
    var manyG = "gggggggggg"
    fmt.Printf("Number of H's in %s is: ", str)
    fmt.Printf("%d\n", strings.Count(str, "H"))         // count occurences
    fmt.Printf("Number of double g's in %s is: ", manyG)
    fmt.Printf("%d\n", strings.Count(manyG, "gg"))      // count occurences
}
```

Counting Occurrences of a Substring in a String

As you can see in the above code, we declare the two strings `str` and `manyG` and initialize them with **Hello, how is it going, Hugo?** and **gggggggggg** at **line 8** and **line 9**, respectively. At **line 11**, we find the count of **H** in `str` using the `Count` function that gives **2** as a result. Similarly, at **line 13**, we find the count of **gg** in `manyG` using the `Count` function, which is **5**.

Repeating a string

The `Repeat` function returns a new string consisting of `count` copies of the string `s`:

```
strings.Repeat(s, count int) string
```

The following program is the implementation of `Repeat` function:

```
package main
import (
    "fmt"
    "strings"
)

func main() {
    var origS string = "Hi there! "
    var newS string
    newS = strings.Repeat(origS, 3)     // Repeating origS 3 times
    fmt.Printf("The new repeated string is: %s\n", newS)
}
```

Repeating a String

As you can see in the above code, we declare a string `origS` and initialize it with **Hi there!** at **line 8**. At **line 10**, we repeat `origS` **3** times using the `Repeat` function and store the new string in `newS` at **line 10**. Then, the result is printed at **line 11**.

Changing the case of a string

The `ToLower` function returns a *copy* of the string `s` with all *Unicode* letters mapped to their *lower case*:

```
strings.ToLower(s) string
```

All uppercase is obtained with:

```
strings.ToUpper(s) string
```

Run the following program to see how these functions work.

```go
package main
import (
    "fmt"
    "strings"
)

func main() {
    var orig string = "Hey, how are you George?"
    var lower string
    var upper string
    fmt.Printf("The original string is: %s\n", orig)
    lower = strings.ToLower(orig)     // changing to lower case
    fmt.Printf("The lowercase string is: %s\n", lower)
    upper = strings.ToUpper(orig)     // changing to upper case
    fmt.Printf("The uppercase string is: %s\n", upper)
}
```

Changing Case of Strings

As you can see in the above code, we declare a string `orig` and initialize it with **Hey, how are you George?** , at **line 8**. At **line 12**, we change the case of `orig` to lowercase, using the function `ToLower`, and at **line 14**, we change the case of `orig` to uppercase, using the function `ToUpper`. We print the new strings at **line 13** and **line 15** to verify the result.

Trimming a string

`TrimSpace` can be used to remove all leading and trailing whitespaces as:

```go
strings.TrimSpace(s)
```

If you want to trim a specific string `str` from a string `s`, use:

```go
strings.Trim(s, str)
```

For example:

```go
strings.Trim(s, "\r\n")
```

The above statement will remove all *leading* and *trailing* **\r** and **\n** from the string `s`. The 2nd string-parameter can contain any characters, which are all removed from the left and right-side of `s`.

If you want to remove only the leading or only trailing characters or strings, use `TrimLeft` or `TrimRight` independently.

Splitting a string

On whitespaces

The `strings.Fields(s)` splits the string `s` around each instance of one or more consecutive white space characters, and returns a slice of substrings []string of `s` or an empty list, if `s` contains only white space.

On a separator

The `strings.Split(s, sep)` works the same as `Fields`, but splits around `sep`. The `sep` can be a separator character (`:`, `;`, `,`, `-`,...) or any separator string `sep`.

Joining over a slice

The `strings.Join(sl []string, sep string)` results in a string containing all the elements of the slice `sl`, separated by `sep`:

Reading from a string

The `strings` package also has a function called `strings.NewReader(str)`. This produces a *pointer* to a *Reader value*, that provides amongst others the following functions to operate on str:

- `Read()` to read a []byte
- `ReadByte()` to read the next byte from the string.
- `ReadRune()` to read the next rune from the string.

Conversion to and from a string

Package *strconv* contains a few variables to calculate the size in bits of the int of the platform on which the program runs:

```
strconv.IntSize
```

Converting a variable of a certain type to a string will always succeed. For converting from numbers, we have the following functions:

```
strconv.Itoa(i int) string
```

It returns the decimal string representation of `i`. Next, we have:

```
strconv.FormatFloat(f float64, fmt byte, prec int, bitSize int) string
```

It converts the 64-bit floating-point number `f` to a string, according to the format fmt (can be 'b','e', 'f' or 'g'), precision `prec`, with `bitSize` being **32** for float32 or **64** for float64.

For converting to numbers, we have the following functions:

```
strconv.Atoi(s string) (i int, err error)
```

It converts to an int. Second, we have:

```
strconv.ParseFloat(s string, bitSize int) (f float64, err error)
```

It converts to a **64**-bit floating-point number

As can be seen from the return-type these functions will return 2 values: the converted value (if possible) and the possible error. So, when calling such a function, the multiple assignment form will be used:

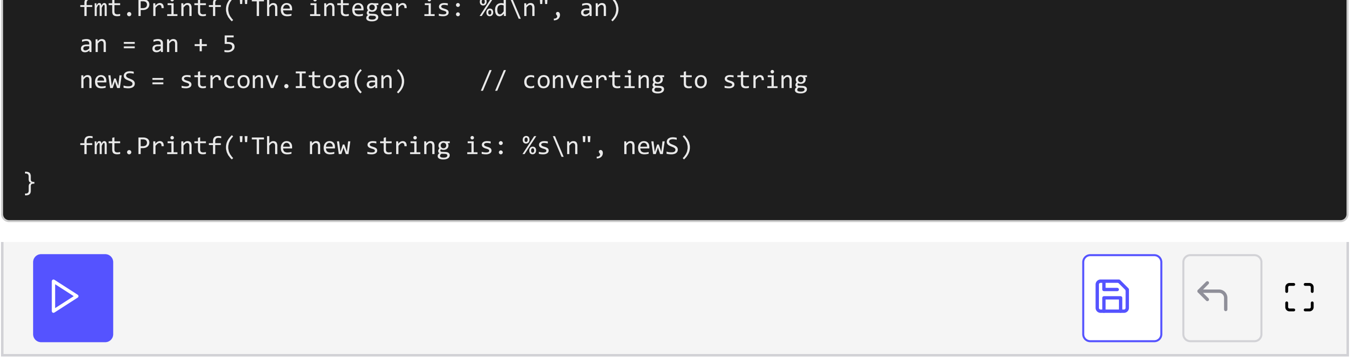
```
val, err = strconv.Atoi(s)
```

Converting a string to another type will not always be possible (as in the case of the functions Atoi and ParseFloat above). Therefore, a *runtime* error is thrown: `parsing "...": invalid argument`.

Run the following programs to see how conversions work:

```
package main
import (
    "fmt"
    "strconv"
)

func main() {
    var orig string = "666"
    var an int
    var newS string
    fmt.Printf("The size of ints is: %d\n", strconv.IntSize)
    an, _ = strconv.Atoi(orig)  // converting to number
```



```
    fmt.Printf("The integer is: %d\n", an)
    an = an + 5
    newS = strconv.Itoa(an)     // converting to string

    fmt.Printf("The new string is: %s\n", newS)
}
```

String Conversion

As you can see in the above code, we declare a string `orig` and initialize it with **666** , at **line 8**. At **line 11**, we calculate the size of ints using `strconv.IntSize` that appears to be **64**. At **line 12** we convert `orig` to an integer using the function `strconv.Atoi(orig)` and store it in a new variable `an`. We modify `an` and convert it into a string at **line 15** using `strconv.Itoa(an)` and then store it in a new variable `newS`. `an` and `newS` are printed at **line 13** and **line 16** to verify the results.

Documentation for other functions in this package can be found here.

That's it about the `strings` and `strconv` package. In the next lesson, you'll study `time` package provided by Golang.