Learning Objectives

Learners will be able to…

- **Work with strings using the “strings” package**

- **Work with files**

info

Make Sure You Know

Python or other modern programming language.

“strings” package

One of th most popular standard Go libraries is the `strings` package.

The package must be imported before use:

```go
import "strings"
```

Mapping Strings

The `ToUpper` and `ToLower` functions are available to change the case of strings. They return strings by converting all letters to uppercase or lowercase. Any characters that are not letters will not be changed.

```go
str := "Hello World"
upper := strings.ToUpper(str)
lower := strings.ToLower(str)
fmt.Println(upper) // HELLO WORLD
fmt.Println(lower) // hello world
```

Trimming white space

`TrimSpace` - returns a string by removing spaces at the beginning and end of the string.

```go
str := " Hello world "
trimSpace := strings.TrimSpace(str)
fmt.Println(trimSpace) // Hello world
```

`Trim`, `TrimLeft` and `TrimRight` - returns a string by removing characters at the beginning and end of the line, on the left or right, respectively.

```go
str = "--Hello world--"
trim := strings.Trim(str, "-")
trimLeft := strings.TrimLeft(str, "-")
trimRight := strings.TrimRight(str, "-")
fmt.Println(trim) // Hello world
fmt.Println(trimLeft) // Hello world--
fmt.Println(trimRight) // --Hello world
```

Trimming custom pre and post fixes

Please note that the characters included in the second parameter of the function (cutset) are removed.

```
str := "--Hello world-a-"
trim := strings.Trim(str, "a-")
fmt.Println(trim) // Hello world
```

`TrimPrefix`, `TrimSuffix` - returns a string without the specified string at the beginning or end, respectively.
Note that we get `prefixHello world` in the output because the prefix substring was removed once.

```
str = "prefixprefixHello world"
trimPrefix := strings.TrimPrefix(str, "prefix")
fmt.Println(trimPrefix) // prefixHello world
```

`TrimFunc`, `TrimLeftFunc`, `TrimRightFunc` - returns a string by removing characters at the beginning and end of the line, on the left or right, respectively. Takes a string and a function that takes an argument of type `rune` and returns a boolean value. Characters that meet the condition will be deleted.
In the example below, characters that are numbers will be removed.

```
str = "Hello world123"
trimFunc := strings.TrimFunc(str, myTrimFunc)
fmt.Println(trimFunc)

// ...

func myTrimFunc(r rune) bool {
    return unicode.IsNumber(r)
}
```

Join, Split, and ReplaceAll

`Join` - joins the elements of a string slice into one line using a separator.

```
strSlice := []string{"one", "two", "three"}
str := strings.Join(strSlice, ", ")
fmt.Println(str) // one, two, three
```

`Split` - works opposite to the `Join` function, splits a string using a separator and returns a slice of strings.

```
str := "one, two, three"
strSlice := strings.Split(str, ", ")
fmt.Println(strSlice) // [one two three]
```

`ReplaceAll` - ReplaceAll returns a copy of the string s with all non-overlapping instances of old replaced by new.

```
str := "one, two, three"
replaceAll := strings.ReplaceAll(str, "one", "two")
fmt.Println(replaceAll) // two, two, three
```

Searching

`HasPrefix` - searches the string from the beginning.
`HasSuffix` - searches the string from the end.
`Contains` - searches anywhere in a string.
`Count` - counts the number of times this string occurs.

```
str := "one, two, three"
replaceAll := strings.Contains(str, "two") // true
count := strings.Count(str, "o") // 2
```

Creating and opening files

To work with files in Go, we can use the `os` package. All files in Go are represented by the `os.File` type.

To create a file we will use the `Create` function. When creating a file, you must pass the path to the file being created.
If the file already exists, it is truncated. If the file does not exist, it is created with mode `0666`(before umask).
If successful, methods on the returned File can be used for I/O; the associated file descriptor has mode O_RDWR.
If there is an error, it will be of type *PathError.

```go
file, err := os.Create("new_file.txt")
if err != nil {
    fmt.Printf("create file error: %s", err.Error())
    return
}
defer file.Close()
```

After finishing working with the file, it should be closed using the `Close` method.

Another way to create and open a file is `OpenFile`.

```go
file, err := os.OpenFile("new_file.txt", os.O_CREATE|os.O_RDWR,
        0644)
if err != nil {
    fmt.Printf("create file error: %s", err.Error())
    return
}
defer file.Close()
```

Here, in addition to the path to the file, we specify specified flags (`O_CREATE`, `O_RDWR`, `O_TRUNC`, etc.).
Where `O_CREATE` indicates that if the file does not exist it will be created.
`O_RDWR` - the file will be opened in read-write mode.
`O_TRUNC` - truncate regular writable file when opened.

Mode perm (before umask) is also indicated. For example `0644`.

info

File information

Among other things, we can find out additional information about the file. To do this, you need to use the `Stat` method.
After which we can find out, for example, the file name, size, mode perm and much more.

```
info, _ := file.Stat()
fmt.Println(info.Name())
fmt.Println(info.Size())
fmt.Println(info.Mode())
```

You can find out more by reading the documentation. It’s also worth paying attention to the `path/filepath` package, which will be useful to you when working with the file system.

Write to file

Once we have created or opened a file, we can start reading or writing data to it.
To write to a file, it has the `Write` method available (implementation of the `io.Writer` interface). The method accepts an array of bytes.
`Write` writes len(b) bytes from b to the File. It returns the number of bytes written and an error, if any. Write returns a non-nil error when n != len(b).

```go
file, err := os.OpenFile("new_file.txt", os.O_CREATE|os.O_RDWR,
        0644)
if err != nil {
    fmt.Printf("create file error: %s", err.Error())
    return
}
defer file.Close()

data := []byte("hello world")
file.Write(data)
```

Also, if we need to write string data, we can simply use the `WriteString` method.

```go
file, err := os.OpenFile("new_file.txt", os.O_CREATE|os.O_RDWR,
        0644)
if err != nil {
    fmt.Printf("create file error: %s", err.Error())
    return
}
defer file.Close()

file.WriteString("hello world")
```

Reading files

To read files, we can use the `ReadAll` function from the `io` package by passing our file to it. An array of bytes is returned.
ReadAll reads from r until an error or EOF and returns the data it read. A successful call returns err == nil, not err == EOF. Because ReadAll is defined to read from src until EOF, it does not treat an EOF from Read as an error to be reported.

```
file, err := os.OpenFile("new_file.txt", os.O_CREATE|os.O_RDWR,
        0644)
if err != nil {
    fmt.Printf("create file error: %s", err.Error())
    return
}
defer file.Close()

data, err := io.ReadAll(file)
if err != nil {
    fmt.Printf("read file error: %s", err.Error())
    return
}
```

We can also use the `Read` method of our file. To do this, we first define a byte slice. In our case we made size 8, although in practice larger should be used. Then, in an endless loop, we will read the data piece by piece until we reach the end of the file, which will be indicated by the error `io.EOF`.

```go
file, err := os.OpenFile("new_file.txt", os.O_CREATE|os.O_RDWR,
        0644)
if err != nil {
    fmt.Printf("create file error: %s", err.Error())
    return
}
defer file.Close()

data := make([]byte, 8)

for{
        n, err := file.Read(data)
        if err == io.EOF{
                break
        }
        fmt.Println(string(data[:n]))
}
```