

Scanning Inputs

This lesson introduces a new method of reading input from the user, i.e., scanning the data from the user.

We already saw how to read from input. Another way to get input is by using the `Scanner` type from the `bufio` package:

Environment Variables

Key:	Value:
GOROOT	/usr/local/go
GOPATH	//root/usr/local/go/src
PATH	//root/usr/local/go/src/bin:/usr/local/go...

```
package main
import (
    "bufio"
    "fmt"
    "os"
)

func main() {
    scanner := bufio.NewScanner(os.Stdin)
    fmt.Println("Please enters some input: ")
    if scanner.Scan() {
        fmt.Println("The input was", scanner.Text())
    }
}
```

Click the **RUN** button, and wait for the terminal to start. Type `go run main.go` and press ENTER.


At **line 9**, we construct a `Scanner` instance from package `bufio` with input coming from the keyboard (`os.Stdin`). At **line 11**, `scanner.Scan()` takes any input (until ENTER key is pressed) and stores it into its `Text()` property, which is printed out at **line 12**. `Scan()` returns true when it gets input, and returns false when the scan stops, either by reaching the end of the input or an error. So, at the end of input, the if-statement is finished.

After the prompt **"Please enter some input: "**, the program waits on `Scan()`

After the prompt "Please enter some input: ", the program waits on `Scan()` for input. If, for example, a text **"Hi there"** is entered, it is read in with the

`Text()` function, and the program outputs: **The input was: Hi there.** The `Scan()` returns false when the scan stops, either by reaching the end of the input or an error.

The `bufio` package also has a `Split` function, which can work at the rune, word or line-level:

Environment Variables 

Key:	Value:
GOROOT	/usr/local/go
GOPATH	//root/usr/local/go/src
PATH	//root/usr/local/go/src/bin:/usr/local/go...

```
package main
import (
    "bufio"
    "fmt"
    "os"
)

func main() {
    scanner := bufio.NewScanner(os.Stdin)
    fmt.Println("Please enter some input: ")
    // scanner.Split(bufio.ScanRunes) // <-- scan runes (newline included)
    scanner.Split(bufio.ScanWords) // <-- scan words (sequence of characters delimited by space)
    // scanner.Split(bufio.ScanLines) // <-- the default: scan lines
    for scanner.Scan(){ // The for loop stops when a EOF is given
        fmt.Printf("Token ->%s<-\\n", scanner.Text())
    }
}
```

Click the **RUN** button, and wait for the terminal to start. Type `go run main.go` and press ENTER.

At **line 9**, we construct a `Scanner` instance from package `bufio` with input coming from the keyboard (`os.Stdin`). Code shows the different ways the `Split` method can be used:

- **Line 11** shows splitting on UTF8 characters (runes).
- **Line 12** shows splitting on spaces, getting the individual words.
- **Line 13** shows splitting on lines (the default).

Here, we used the second option at **line 12**. At **line 14**, the `Scan()` method is

called. As we know, `Scan()` returns true when it gets input, and returns false

when the scan stops, either by reaching the end of the input or an error. Put inside an infinite for-loop (from **line 14** to **line 16**), this executes by printing out the input found until the end of input or an error condition is reached.

We can customize the splitting. For example, adapt it, so that split words are seen as a sequence of alphanumeric characters only; so " **Hello, how are you?**" would be split in six tokens.

Environment Variables



Key:	Value:
GOROOT	/usr/local/go
GOPATH	//root/usr/local/go/src
PATH	//root/usr/local/go/src/bin:/usr/local/go...

```
package main
import (
    "bufio"
    "fmt"
    "os"
    "unicode"
    "unicode/utf8"
)

func main() {
    scanner := bufio.NewScanner(os.Stdin)
    fmt.Println("Please enter some input: ")

    // scanner.Split(bufio.ScanWords)
    scanner.Split(AlphanumericWord)
    for scanner.Scan() {
        fmt.Printf("Token ->%s<-\n", scanner.Text())
    }
}

func AlphanumericWord(data []byte, atEOF bool) (int, []byte, error) {
    // Skip leading spaces
    start := 0
    for width := 0; start < len(data); start += width {
        var r rune
        r, width = utf8.DecodeRune(data[start:]) // Read a rune
        if !unicode.IsSpace(r) {
            break // Break if a non-space character has been found (beginning of a word)
        }
    }
    // Start reading a word
    for width, i := 0, start; i < len(data); i += width {
        var r rune
        r, width = utf8.DecodeRune(data[i:]) // Read a rune
        // Stop when you read something that is not alphanumeric
        if !(unicode.IsLetter(r) || unicode.IsDigit(r)) {
```

```

17: if !unicode.IsLetter(r) || !unicode.IsDigit(r) {
18:     if i == 0 {
19:         // if the first character is not a space (already skipped)
20:         // but it is not an alphanumeric character, print that character alone
21:         return i + width, data[start : i+width], nil
22:     } else {
23:         // Otherwise you just completed a word
24:         return i, data[start:i], nil
25:     }
26: }
27: }
28: if atEOF && len(data) > 0 {
29:     return len(data), data[0:], nil
30: }
31: return 0, nil, nil
32: }

```

Click the **RUN** button, and wait for the terminal to start. Type `go run main.go` and press ENTER.

This program has the exact same structure as the previous code, except **line 15**, where `Split()` is called. It is now called with a function `AlphanumericWord` as an argument, which is defined from **line 21** to **line 51**. This is the way to customize our `Split()`: the input will be stored in the variable `data` which is of type `[]byte`.

Now, let's see the implementation of `AlphanumericWord()`. The major part is about skipping *initial* spaces getting to the first word. It goes through the data with a for-loop starting at **line 24**. Then, at **line 27**, when the first *non-space* character is reached (`!unicode.IsSpace(r)`), the loop stops with a `break`. Then, we start a for-loop at **line 32** to read that word: from the `i` position we were on (`i = start`, see **line 32**), we read a character (rune) `r`. If `r` is not a letter or not a digit (**line 36**) and we are not at the first character (else at **line 41**), we return a complete word `data[start:i]` that we found at **line 43**. This continues until `i >= len(data)`. Then, we are at the end of the input at `EOF` (see **line 47**), and the remaining input text is returned (see **line 48**).

Now that you are familiar with the different concepts of reading and writing, how about designing interfaces for such purposes? In the next lesson, you will learn building interfaces for reading and writing purposes.