```swift
import UIKit

import AppDeveloperKit_String
```

Contents

Overview

AppDeveloperKit-String is a Swift String library that provides support for subscripting and easy to use regular expression matching and substitutions.

The matching and substitution support was inspired by Perl and the syntax was developed to resemble that language.

- Use of the =~ operator to mimic Perl's =~ binding operator.
- A tuple for the right hand side to allow multiple parameters (pattern, replacement, flags, etc).

Compare the examples in the following section that show Perl and AppDeveloperKit-String side by side.

Quick Introduction

```
// Perl
// my ($d1,$d2) = "A56B" =~ m/(\d)(\d)/;    // ("5","6")
// $` = "A"
// $& = "56"
// $' = "B"

var (d1,d2) = "A56B" =~ (.M,"(\\d)(\\d)", { (preMatch, match,
 postMatch) in  // ("5","6")
    preMatch // A
    match    // 56
    postMatch // B
})



// Perl
// my @arr = "A56B" =~ m/(\d)(\d)/;    // ["5","6"]

var arr1: [String] = "A56B" =~ (.M,"(\\d)(\\d)")   // ["5", "6"]



// Perl
// my $str = "WX YZ";
// my $count = $str =~ s/(\w)(\w)/$2$1/g;   // $count = 2, $str = "XW
 ZY"

var str = "WX YZ"
var count = str =~ (.S,"(\\w)(\\w)","$2$1","g") // count = 2, str =
 "XW ZY"
```

Substitutions

Syntax

> Example
>
> ```
> var str: String
>
> str =~ (.S, <pattern>, <replacement> [, <flags>])
> ```

Pattern

The pattern can consist of whatever is supported by NSRegular Expression. See the table for Regular Expression Metacharacters

The ICU regular expressions supported by NSRegularExpression are described in the ICU User Guide

Replacement

The replacement text can include a template as supported by NSRegularExpression.

The template specifies what is to be used to replace each match, with $0 representing the contents of the overall matched range, $1 representing the contents of the first capture group, and so on.

Flags

The flags argument is optional. Flags may be appear in any order. Specifying a flag more than once is equivalent to specifying it once. Unrecognized flags are ignored.

Supported flags:

- g Global replacement
- i Case insensitive replacement

Using wrapper class

> Example
>
> ```
> Regex.s(str: &str, pattern: <pattern>, replacement: <replacement>, flags:
> <flags>)
> ```

Examples

```
// In-place substitution using a pattern and replacement with
 template.  Returns count of substitutions made.
str = "XY"
count = str =~ (.S,"(\\w)(\\w)","$2$1") // 1
str // YX


// Variant using a flags argument.
str = "XY ZW"
count = str =~ (.S, "(\\w)(\\w)","$2$1", "g") // 2
str // YX WZ


// An alternate interface using the Regex class.
str = "XY"
count = Regex.s(str: &str, pattern: "([a-z])([a-z])", replacement:
 "$2$1", flags: "i") // 1
str // YX
```

Matching - Tuple result

Syntax

Tuple result

One to ten variables may be returned in a tuple from the output of match.

> Example
>
> ```
> var (v1 [[, v2] ... [, v10]]) = <str> =~ (.M, <pattern> [, <flags>])
> ```

Tuple result with closure

The closure provides access to pre-match, match and post-match results as optional variables.

> Example
>
> ```
> var (v1 [[, v2] ... [, v10]]) = <str> =~ (.M, <pattern> [, <flags>],
> { (<preMatch>, <match>, <postMatch>) in
>    <code>
> })
> ```

Tuple notes

- See definition of pattern and flags in Substitution section.
- The output of a match will obey the following rules:
    - If pattern contains at least one capture group that participates in matching, then all capture groups are returned as output.
    - If pattern contains no participating capture groups and the global flag is used, then all matches are returned as output.
    - If pattern contains no participating capture groups and the global flag is not used, then all variables in tuple will be nil.
    - If more variables exist in tuple than there are capture groups or matches (as relevant), then excess variables will be nil.
- Closure will obey the following rules:
    - If there is no match, all three closure variables will be nil (prematch, match postmatch).

Using wrapper class

> Example
>
> ```
> var (v1 [[, v2] ... [, v10]]) = Regex.m(str: <str>, pattern: <pattern>,
> flags: <flags>, completion: { (preMatch, match, postMatch) in
>    <code>
> ```

```
// A tuple with from 1 - 10 arguments supported
str = "0123456789"
var (a,b,c,d,e,f,g,h,i,j) = str =~ (.M,"(\\w)(\\w)")
(a,b,c,d,e,f,g,h,i,j) // (0, 1, nil, ... nil)




// With Flags argument
str = "XY"
(a,b) = str =~ (.M,"([a-z])([a-z])", "i")  // ("X", "Y")




// Completion block additionally provides preMatch, match and
 postMatch results
// Only grab the capture groups you need.
str = "4XY5"
(a) = str =~ (.M,"([A-Z])([A-Z])", { (preMatch, match, postMatch)
 in // ("X")
    preMatch // 4
    match    // XY
    postMatch // 5
})



// With flags support
str = "4XY5"
(a) = str =~ (.M,"([a-z])([a-z])", "i", { (preMatch, match, postMatch)
 in  // ("X")
    preMatch // 4
    match    // XY
    postMatch // 5
})
```

```
// Global flag example with capture groups.
str = "-12 34 56-"
(a,b,c,d,e,f) = str =~ (.M,"(\\d)(\\d)", "g", { (preMatch, match,
 postMatch) in // ("1","2","3","4","5","6")
    preMatch // -12 34
    match    // 56
    postMatch // -
})


// Global flag without capture groups will return the matches.
str = "-123456-"
(a,b,c) = str =~ (.M,"\\d\\d", "g", { (preMatch, match, postMatch)
 in // ("12","34","56")
    preMatch // -1234
    match    // 56
    postMatch // -
})


// Pattern contains no participating capture groups and the global
 flag is not used, so all variables in tuple will be nil.
str = "-12-"
(a,b) = str =~ (.M,"\\d\\d", { (preMatch, match, postMatch) in //
 (nil,nil)
    preMatch // -
    match    // 12
    postMatch // -
})
```

```
// The capture group (a) does not participate, so we still have no
 participating capture groups. Since global
// flag is not used, a is nil.
str = "4"
(a) = str =~ (.M,"(a)|\\S", { (preMatch, match, postMatch) in // (nil)
    preMatch // –
    match    // 4
    postMatch // –
})



// No match will produce nil for all 3 variables in closure.
str = ""
(a) = str =~ (.M,"\\S", { (preMatch, match, postMatch) in // nil
    preMatch // nil
    match    // nil
    postMatch // nil
})



// A match can consist of an empty string
str = ""
(a) = str =~ (.M,"\\S*", { (preMatch, match, postMatch) in // nil
    preMatch // ""
    match    // ""
    postMatch // ""
})




// An alternate interface using the Regex class.
str = "4XY5"
(a,b) = Regex.m(str: str, pattern: "([a-z])([a-z])", flags: "i",
 completion: { (preMatch, match, postMatch) in // ("X","Y")
```

```
    preMatch // 4
    match    // XY
    postMatch // 5
})
```

Matching - Array result

Syntax

Array result

Output of match returned as an array - either optional or non-optional.

When the optional form is used, one can distinguish between no match (nil result) versus an empty result (no capture groups or matches returned as array).

> Example
>
> ```
> var <arr>: [String] = <str> =~ (.M, <pattern> [, <flags>])
> var <arr>: [String]? = <str> =~ (.M, <pattern> [, <flags>])
> ```

Array result with closure

The closure provides access to pre-match, match and post-match results as optional variables.

> Example
>
> ```
> var <arr>: [String] = <str> =~ (.M, <pattern> [, <flags>], { (<preMatch>,
> <match>, <postMatch>) in
>    <code>
> })
>
> var <arr>: [String]? = <str> =~ (.M, <pattern> [, <flags>],
> { (<preMatch>, <match>, <postMatch>) in
>    <code>
> })
> ```

Array notes

- See definition of pattern and flags in Substitution section.
- The output of a match will obey the following rules:
  - If pattern contains at least one capture group that participates in matching, then all capture groups are returned as array.
  - If pattern contains no participating capture groups and the global flag is used, then all matches are returned as array.
  - If pattern contains no participating capture groups and the global flag is not used, the array will be empty.
- If there is no match:
  - If array is specified as optional, it will be nil.

```
// A nil result with optional array signifies no matches.
str = "XY"
var arrOpt: [String]? = str =~ (.M,"(\\d)") // nil


// Capture group returned in array
str = "XY"
arrOpt = str =~ (.M,"(\\w)") // ["X"]


// Matches returned in array when there are no participating capture
 groups AND global flag used.
str = "-123456-"
arrOpt = str =~ (.M,"\\d\\d", "g", { (preMatch, match, postMatch)
 in // ("12","34","56")
    preMatch // -1234
    match    // 56
    postMatch // -
})


// The capture group (a) does not participate, so we still have no
 participating capture groups. Since global
// flag is not used, arrOpt is [].
str = "4"
arrOpt = str =~ (.M,"(a)|\\S", { (preMatch, match, postMatch) in // []
    preMatch // -
    match    // 4
    postMatch // -
})


// Take your pick: Optional or non-optional array return value.
// Here we can't determine if there was a match.   If we need this
 information with non-optional array
// then use the closure form.
str = "XY"
var arr: [String] = str =~ (.M,"(\\d)") // []


// With or without flags
str = "XY"
arr = str =~ (.M,"([a-z])", "ig") // ["X", "Y"]


// Completion handler available.
str = "4XY5"
arr = str =~ (.M,"([a-z])", "gi", { (preMatch, match, postMatch)
 in   //  ["X","Y"]
    preMatch // 4
    match    // Y
    postMatch // 5
})


// Array will be empty without a match, but the closure variables will
 be nil.
str = ""
arr = str =~ (.M,"\\S", { (preMatch, match, postMatch) in // []
    preMatch // nil
    match    // nil
    postMatch // nil
})


// An alternate interface using the Regex class.
str = "4XY5"
```

```
arr = Regex.m(str: str, pattern: "([a-z])", flags: "ig", completion: {
 (preMatch, match, postMatch) in   //  ["X","Y"]
    preMatch // 4
    match    // Y
    postMatch // 5
})
```

Matching - Bool result

Syntax

Bool result

Output of match returned as Bool.

true - Found a match.

false - No match.

> Example
>
> `var <result>: Bool = str =~ (.M, <pattern> [, <flags>])`

Bool notes

- See definition of pattern and flags in Substitution section.

Using wrapper class

> Example
>
> `var <result>: Bool = Regex.m(str: <str>, pattern: <pattern>, flags: <flags>)`

Examples

```
str = "XY"
var result: Bool = str =~ (.M,"\\d") // false
```

```
// With flags
result = str =~ (.M,"[a-z]", "i")  // true


// An alternate interface using the Regex class.
result = Regex.m(str: str, pattern: "[a-z]", flags: "i") // true
```

Special Cases

```
// Unicode characters are supported.

// Need to use a character expression that matches a grapheme cluster
str = "😎"  // "\u{1F60E}"
result = str =~ (.M,"(\\X)")  // true


// A ZWJ sequence needs special handling.  A single \X will not match.
str = "👩‍🚀"  // "\u{01F469}\u{200D}\u{01F680}"
result = str =~ (.M,"(\\X)")  // false


// Instead we need to account for a possible Zero Width Joiner (ZWJ)
 code point \u{200D}
// Here is a nice relevant article: http://cyrilwei.me/swift/
 2016/08/31/the-emoji.html
//
result = str =~ (.M,"(\\X(?:\u{200D}\\X)*)")  // true
```

String subscripting

```
// Subscripting by index, Range and ClosedRange.
// Unicode supported.

str = "A123CDE"

str[1] = "😎" // Assignment supported
str[2..<4] =  "B🧑‍🚀"  // Including with ranges

str[1] // 😎
str[2] // B
str[3] // 🧑‍🚀

str[1..<4] // 😎B🧑‍🚀

str[1...4] // 😎B🧑‍🚀C
```