Swift Programming
Lecture 8

String, Tuple, Set, and Dictionary

Tatsuo Minohara

文字列の入力

- 文字端末から
  - 文字列変数? = readLine( )
- オプション型で返される。nilを仮定しないときは、ラッパを外す!演算子が必要になる

```
var s = readLine( )!
if  let s = readLine( ) {
    // 文字列が入力されたときにすること

}
```

文字列の作成

- 文字列は、リテラルでも表現できるが、コンストラクタでも作成できる
- 書式： String(値)

  String(234) → "234"

  String(true) → "true"

  String(56.7e12) → "56700000000000.0"

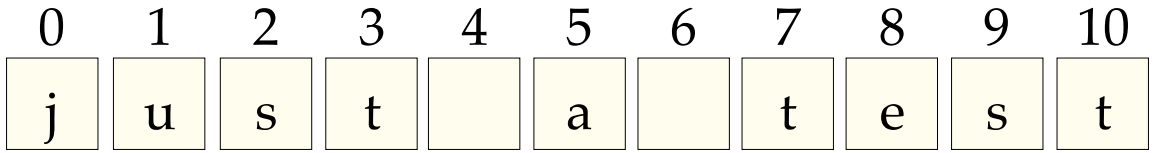
- 書式： String(repeating: 文字列, count: 個数)

  String(repeating: "ABC", count: 3) → "ABCABCABC"

  String(repeating: "あ", count: 5) → "あああああ"

文字の位置

- 次に出てくるリストと同じで、0文字目から始まる。
- 文字列の順番は、「0〜文字列の長さ-1」の範囲で振られている。



文字列.startIndex

文字列.endIndex

インデックスの型は、String.Index

文字列の各文字を取り出す

- 文字列の長さを求める
  - s.count　// sに代入されている文字列の長さを返す
- 文字列の各文字を取り出すには、一度配列に変換する必要がある
  - var sa = Array("ABCEDF")　// ["A", "B", "C", "D", "E", "F"]
  - saの型は、[String.Element] になる。String.Element（あるいはCharacter）のリスト
- 1 文字の配列は、Stringにすることによって、文字列に戻る
  - String(sa) // "ABCDEF" という
- 配列の要素として 1 文字を取り出すときは、インデックスとして文字列インデックス型（String.Index）を用いる
  - 文字列[String.Index(utf16Offset: インデックス , in: 文字列)] → String.Element型の値が返される
  - インデックスは整数式で、範囲は0〜文字列の長さ-1
  - 例：

```
s[ String.Index( utf16Offset: 0, in: s ) ]        s[ String.Index( utf16Offset: 3, in: s ) ]
s[ String.Index( utf16Offset: 5, in: s ) ]        s[ String.Index( utf16Offset: s.count - 2, in: s ) ]
```

文字列の範囲指定

- 文字列の中から範囲を指定して一定の範囲の文字列を取り出す
  - String.Substringという型になるので、文字列に戻すには、String(範囲指定された文字列)を使う
- 範囲指定の記法では、以下の通り、インデックス（String.Index）を用いる
  - 文字列[始め...終わり]
  - 文字列[始め..<終わりの１つ後]
  - 文字列[...終わり] // 始めは0が仮定される
  - 文字列[..<終わりの１つ後] // 始めは0（startIndex）が仮定される
  - 文字列[始め...] // 終わりは文字列.count-1が仮定される
- 例：

```
let s = "ABCDEFGHIJ"
String( s[ String.Index( utf16Offset: 1, in: s )...String.Index( utf16Offset: 5, in: s ) ] )   // "BCDEF" の５文字
String( s[ String.Index( utf16Offset: 1, in: s )..<String.Index( utf16Offset: 5, in: s ) ] )   // "BCDE" の４文字
String( s[ s.startIndex..<s.endIndex ] ) // すべての文字 "ABCDEFGHIJ"
String( s[ String.Index( utf16Offset: 5, in: s )... ] )   // "FGHIJ" ６番目から最後まで
String( s[ ...String.Index( utf16Offset: 3, in: s ) ] )   // "ABCD" 最初の４文字
```

部分文字列の指定

- 文字列.subscript(範囲指定) → 部分文字列が返される
    - 使い方は、文字列.[範囲指定]とだいたい同じ
- 文字列.prefix(整数) → 先頭から、指定された文字数分だけ返す
- 文字列.prefix(through: 文字列インデックス) → 先頭から、指定された文字列インデックスの文字まで返す
- 文字列.prefix(upto: 文字列インデックス) → 先頭から、指定された文字列インデックスの文字の前まで返す
- 文字列.suffix(整数) → 最後から、指定された文字数分だけ返す
- 文字列.prefix(from: 文字列インデックス) → 指定された文字列インデックスの文字から最後の文字までを返す
- 文字列.dropFirst(整数) → 先頭から指定文字数を除いた残りの部分文字列を返す
- 文字列.dropLast(整数) → 最後から指定文字数を除いた残りの部分文字列を返す

文字列インデックスの制御

- 文字列.index(after: 文字列インデックス) … 次の文字列インデックスを返す
- 文字列.index(before: 文字列インデックス) … １つ前の文字列インデックスを返す
- 文字列.formIndex(after: &文字列インデックス) …その文字列インデックスを１つ後ろにする
- 文字列.formIndex(before: &文字列インデックス) …その文字列インデックスを１つ前にする
- 文字列.index(文字列インデックス, offsetBy: 整数) … 文字列インデックスに整数を加えた文字列インデックスを返す
- 文字列.formindex(文字列インデックス, offsetBy: 整数) … その文字列インデックスに整数を加える
- 文字列.distance(from: 文字列インデックス, to: 文字列インデックス) … ２つの文字列インデックス間の距離を求める

文字列から文字を取り出す

- 文字列.subscript(文字列インデックス) … 指定されたインデックスの文字を取り出す
- 文字列.first? … 最初の文字（あれば）を取り出す
- 文字列.last? … 最後の文字（あれば）を取り出す
- 文字列.randomElement() … 文字列の中からランダムに 1 文字取り出す
- 繰返しを使っても取り出せる

```
for (i, c) in "ABCDEFG".enumerated() {
    print( i, c )
    // このときのcの型は、String.ElementあるいはCharacter

}
```

文字列の比較、照合など

- **文字列は、==で等しいかどうか比べる**
- 文字列.count …文字列の長さ
- 文字列.contains(検索文字) … その検索文字が入っているかどうか（検索文字は１文字だけ）
- 文字列.contains(検索文字列) … その検索文字列が入っているかどうか（import Foundationが必要）
- 文字列.starts(with: 比べる文字列)…その文字列で始まるか
- 文字列.elementsEqual(比べる文字列)…その文字列と等しいか
- 文字列.hasPrefix(文字列)…その文字列で始まるか
- 文字列.hasSuffix(文字列)…その文字列で終わるか

文字列の検索

- 文字列.firstIndex(of: 検索文字) ? …最初に見つかった文字のインデックスを返す（返される位置は0から、見つからなければnil）
- 文字列.lastIndex(of: 検索文字) ? …最後に見つかった文字のインデックスを返す
- 文字列.range(of: 検索文字列) ? …開始位置から検索、import Foundationが必要、見つかった文字列の範囲（Range<String.Index>)を返す、なかったらnilが返ってくる（結果!.lowerBoundで開始のインデックス、結果!.upperBoundで終了のインデックスが返される）
- ??演算子
- nilだったら、??以降の値が代入される
    - let s = "ABCDEF"
    - let a = s.firstIndex("G") ?? s.endIndex // Gが見つからなければ最後のインデックスを代入

配列と同様の追加、挿入、削除関数

- letで宣言された文字列変数にはできない、varで宣言する必要がある
- 文字列変数.append(文字列) … 後ろに追加する 文字列 += 文字列
- 文字列変数.insert(文字, at: 文字列インデックス)
- 文字列変数.remove(at: 文字列インデックス)
- 文字列変数.removeAll( )
- 文字列変数.removeFirst( )
- 文字列変数.removeFirst(文字数)
- 文字列変数.removeLast( )
- 文字列変数.removeLast(文字数)
- 文字列変数.removeSubrange(文字列の範囲指定)

配列と同様の順番を変える関数

文字列に変更を加えて新たな文字列を作る関数

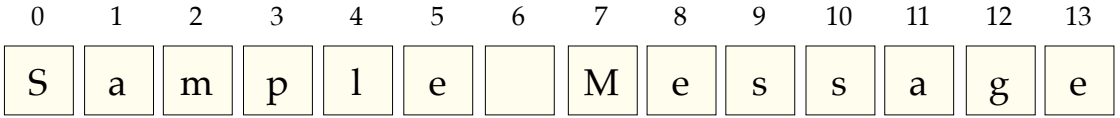
- 文字列.sorted() → 文字の配列ができる
- 文字列.reversed() → 反転された文字列の集合
- 文字列.shuffled() → 順番がばらばらになった文字の配列
- 文字列に戻すには、String( )の中に入れる必要がある

- 文字列.replacingOccurrences(of: "検索文字列", with: "置換文字列") →すべて置換された文字列
- 文字列.trimmingCharacters(in: .whitespaces) →前後の空白が取られた文字列

- 文字列.uppercased( ) → 英大文字に変換された文字列
- 文字列.lowercased( ) → 英小文字に変換された文字列
- 文字列.capitalized(Locale(identifier: "en_US_POSIX")) → 最初の 1 文字を大文字にして、後の文字を小文字にする

文字列の操作

- **typealias** SI = String.Index
- **let** s = "Sample Message"
- **let** r = s.range(of: "Message")　// Range<String.Index> 7 ~ 13
- **let** slen = SI(utf16Offset: s.count, in: s)
- **let** s1 = String(s[SI(utf16Offset: 4, in: s)..<SI(utf16Offset: 9, in: s)]) // "le Me"
- **let** s2 = String(s[s.index(slen, offsetBy: -3)...]) // "age"
- **let** s3 = String(s[...SI(utf16Offset: 2, in: s)])　　// "Sam"
- **let** s4 = s.replacingOccurrences(of: "Mess", with: "her ") // "Sample her age"



各文字の種類を調べる

- cがCharacter（あるいはString.Element）型だと、一度文字列にして、UnicodeScalar型に変換しないといけない
  - **let** us = Unicode.Scalar(String(c))! // optional型なので
- Unicode.Scalar型のオブジェクトのproperties属性については、以下のような属性が用意されている（論理値）が、一部を紹介
  - isASCIIHexDigit, isAlphabetic, isDash, isEmoji,
  - isHexDigit, isIdeographic, isLowercase, isMath,
  - isUppercase, isWhitespace
- 利用例：
  - **if** us.properties.isAlphabetic { print("\(c) is an alphabet.") }

文字列に対する高階関数

- 文字列.filter(関数またはクロージャ)
- 文字列.map(関数またはクロージャ)
- 文字列.flatMap(関数またはクロージャ)
- 文字列.reduce(関数またはクロージャ)

再帰で文字列を

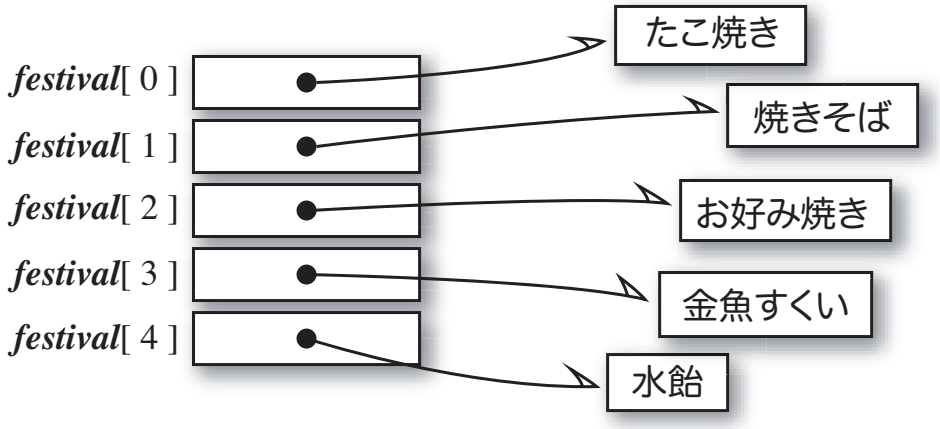
- 回文を作る例
  - 最初は、任意の文字を選ぶ
  - 前後に同じ文字（任意の文字）を追加していく
  - これを再帰で行なう

文字列のリスト

- リストの各要素が文字列になる。
- 空の文字列のリストとして初期化する
  - リスト変数名 =[""] * サイズ
    - 例：var namelist =Array(repeating: "", count: 10)
- 各要素に対して、文字列を代入できる。
    - 例：namelist[0] = "Hello, Python"
    - namelist[3] = namelist[0] + " and Script"

文字列のリストの初期化

- **let** festival = ["たこ焼き", "焼きそば", "お好み焼き", "金魚すくい", "水飴"]



文字列とリストの変換

- Array(文字列)…1 文字ずつ分解したリストができる
- String(リスト)…print関数で表示されるような文字列ができる

- 文字列.split(separator: 区切り文字)…文字列を区切り文字で区切って、リストにする
- 文字列の配列.joined(separator: 区切り文字)…文字列の配列の各要素を区切り文字で繋げて文字列にする

文字列の分割

- stringクラス用のsplitメソッド
- １つの文字列を、区切りとなる文字列で分割し、文字列のリストに直すことができる
  - 文字列.split(separator: 区切りとなる文字列)
  - 例：

```
let tokens = "A sample message".split( separator: " " )
// tokens = String.Subsequence[ "A", "sample", "message" ]
for token in tokens {
        print( token )
}
```

文字列の統合

- 文字列の配列から、間にいれる文字列を指定して、１つの文字列に統合することができる
  - 文字列のリスト.joined(separator: "区切り文字列")
- 例：

```
let colorlist = [ "red", "blue", "green", "white" ]
let total = colorlist.joined( separator: " and ")
// total = "red and blue and green and white"
print( total )
```

正規文字列を使った検索

- 正規文字列を扱うモジュール NSRegularExpression
  - **import** Foundation // 利用する前に
- 正規文字列で検索
  - **let** マッチオブジェクト = **try**! NSRegularExpression(pattern: 正規文字列, option: [])
  - **let** 検索結果オブジェクト = マッチオブジェクト.matches(in:検索対象文字列, options: [], range: NSRange(範囲指定))
  - 例：

```
let source = "345th conference at 2021 year"
let matobj = try! NSRegularExpression(pattern: "[0-9]+" )
let results = matobj.matches( in: source, options: [], range:
NSRange(0..<source.count) )
```

正規文字列検索結果の表示

- 検索結果のオブジェクトを用いて検索結果を表示する
  - 見つからなければnilになっている
  - start, endで、見つかった先頭の位置、終了の位置を求められる
- 例：

```
for result in results {
    for i in 0..<result.numberOfRanges {
        let start = source.index(source.startIndex, offsetBy: result.range(at: i).location)
        let end = source.index(start, offsetBy: result.range(at: i).length)
        let text = String(source[start..<end])
        print(text )
    }
}
```

  - 参考サイト：https://www.2nd-walker.com/2020/04/21/swift-how-to-use-nsregularexpression/

正規文字列

- UNIXの伝統の正規文字列
- .　…　任意の一文字
- ?　…　任意の0文字か１文字
- *　… 0回以上の繰返し
- + … 1回以上の繰返し
- \a　… その文字自体を表わす　\.　\?　\\　*
- [abc]　…　文字のグループ化　[a-z,A-Z,0-9]　[b-x]
- [^abc]　… ^はそれは含めない
- ^ …行の先頭
- $ …行の最後

配列／リスト（復習）

- 複数の値をまとめておける
  - 配列、タプル、辞書、集合がSwiftには用意されている
- 配列の記述は、[ ]を用いる
  - 例：**let** numlist = [2, 3, 5, 7, 11, 13, 17, 19, 23, 31] # 素数リスト
- 配列の個々の値は、要素と呼ばれる
- Swiftでは、要素の型は異なっていても構わない
  - 例：**let** heterogeneous = [345, true, "WOW", 45.6e-12] **as** [Any]
- 配列の要素の個数を調べるのにcount属性が使える
  - 例：numlist.count

インデックス式

- インデックスの式を用いて、要素を１つ取り出すことができる
- 文字列のインデックスと同じで、範囲は、
  正の整数の場合：0〜要素の個数-1
  負の整数の場合：-1 〜 - 要素の個数
- 例
  - numlist[0]　# 最初の要素
  - numlist[-1] # 最後の要素
  - numlist[len(numlist) -1] # 最後の要素

範囲指定式

- 範囲指定で配列の複数の要素を取り出すことができる
- 結果は、Array.Subsequenceのオブジェクトとして返される
- 例：
  - Array(numlist[2...3]) # 1個の要素の場合でも配列として返す
  - Array(numlist[5...9]) # ５番目から９番目の前まで
  - Array(numlist[..<5]) # 最初の５個の要素
  - Array(numlist[numlist.count-5...]) # 最後の５個の要素

- strideが使えないので要注意

範囲式を使った配列への代入

- 配列の場合、範囲式を使って代入ができる（文字列は変更不可能なのでエラーになる）
- 代入後は、元の部分が、代入した配列に置き換えられる。
- 例：

```
a[ 4...6 ] = [ 3, 4, 5, 6, 7 ]
```

配列への変換

- Array(arrayLiteral:)関数を使う
  - 書式：Array(arrayLiteral: 値, 値, …)
  - 例：Array(arrayLiteral: 56, 32, 193, 23)
  - 評価は、[56, 32, 193, 23]となる
- 範囲指定から、配列の形にするときにもArray関数が用いられる
  - 例：Array(1...5)
  - 評価：[1, 2, 3, 4, 5]

タプル

- タプルは変更不可能なデータの連なりで、一般的に異種のデータの集まりを示す
- 丸括弧の対を使う：書式 (値, …)
- ( )は、特殊な空のタプルを表す
- 単要素のタプルはない: (a) は、aと等しい
- 項目をカンマで区切る: a, b, c または (a, b, c)
- タプルは代入や比較は可能（比較は先頭の要素から比較される）
  - 例：var (x, y) = (12, 34)
  - (12, 34) < (78, 89)
- Swiftでは、タプルはクラスとして定義されていない。配列(Arrray)、文字列(String)に共通に用意されている演算（代入・比較演算以外）や関数を適用すると、インタープリタが落ちるか、エラーになるので要注意

タプルの要素へのアクセス

- インデックス式は使えない
- その代わりに、ドット.整数で要素を取り出せる
  - 整数は、正の整数で0から始まる
  - 例：var atuple = (12, 23, 45)

    print(atuple.0, atuple.2 ）// 要素の参照

    atuple.1 = 789　// 代入も可能

キーワード付きタプル

- タプルにキーワードを入れることができる
  - 書式：(キーワード名: 値, …)
- 例：
  - **let** record = (name: "John", age: 34)
  - **var** r:Int, g:Int, b: Int
  - (red: r, green: g, blue: b) = (red: 23, green:34, blue: 45)
  - **var** (r, g, b) = (red: 23, green:34, blue: 45)

- キーワード名が異なると代入や比較ができないので注意
- キーワード付きのタプルとキーワードなしのタプルは、相互に代入や比較が可能

配列・文字列の共通演算

- xを式とし、sおよびtを配列（あるいは文字列）とする
- s.contains(x)
  - xがsに含まれている（いないかどうか）
  - true/falseが返される
- s + t
  - 結合された新しい配列・文字列が返される
- s.count
  - 個数や長さが整数で返される
- s 比較演算子 t　（比較演算子==, !=, <, >, >=, <=など）
  - 最初の要素（文字）から比較される

配列・文字列の演算用の関数

- s.min( ) あるいは s.max( )
  - s中の要素の最小値／最大値が返される
- s.index(x)
  - s中で指定された値が何番目にあるかが返される（ただし、0番始まり）
  - s.index(x, 検索開始のインデックス)
  - s.index(x, 検索開始のインデックス, 検索終了+1のインデックス)
- s.count(x)
  - sの中に何個指定された値があるのか整数で返される

Set（集合型）

- 集合型は、配列と同じように[ ]で要素を囲むが、型をSetに指定する、あるいは配列から、Setコンストラクタで変換する
  - 例：**let** fruits: Set = ["りんご", "みかん", "なし", "いちご"]
  - 例：**let** oddnums: Set<Int> = [1, 3, 7, 9, 13, 17]
  - 例：**let** drinks = Set(["紅茶", "コーヒー" , "ジュース"])
- 集合なので、重複項目は、除去される
  - **let** aset = Set([1, 2, 2, 3, 3, 4])
  - **let** aary = Array(aset)
  - // [4, 3, 2, 1]となる

集合型の値に対する演算・属性

- contains関数を使って、集合に入っているかどうかを判定
  - 例：fruits.contains("なし")
- ==あるいは、!=で、同じ値を持つ集合かどうかを判定
  - 例：fruits == drinks
- count属性で、集合内の要素の個数を得られる
  - 例：fruits.count
- isEmpty属性で、集合に要素があるかどうかを調べることができる
  - 例：fruits.isEmpty

集合型の値に対する関数

- sorted()関数で、要素を昇順などでソートさせた新しい集合を作成
  - 例：fruits.sorted( )
- insert(), remove(), removeAll()で、要素を追加、削除、全削除する
  - 例：fruits.insert("なし")
  - fruits.remove("りんご")
  - fruits.removeAll( )
- for in ループで、要素を順次取り出すことができる
  - 例：**for** f **in** fruits { print(f) }

Setでの演算関数

- 集合同士の演算→結果の集合が返される
  - 和集合 union　A.union(B)
  - 差集合 subtracting　A.subtracting(B)
  - 交差集合 intersection　A.intersection(B)
  - 排他的論理和集合 symmetricDifference　A. symmetricDifference(B)
- 部分集合の判定→論理値が返される
  - 部分集合か　A.isSubset(of: B)
  - 真部分集合か　A.isStrictSubset(of: B)
  - 上位集合か　A.isSubset(of: B)
  - 真上位集合か　A.isStrictSubset(of: B)
  - 共通要素がないか　A.isDisjoint(with: B)

辞書（Dictionary）

- 辞書では、キーでエントリの値を引いてくることができる、ただしキーは一意である必要がある
- 「キー値：対応する値」のペア（エントリ: entryと呼ばれる）を、[ ]の中に入れることで、辞書を作ることができる。空の辞書の場合は、[:]で指定できる
  - 例：**let** flowers = ["rose" : "薔薇", "lily": "百合", "hydrangea": "紫陽花"]
  - 例：**var** numbers : [Int : String] = [:]
- Dictionary生成関数（コンストラクタ）を使っても、空の辞書を作成することができる
  - 例：**var** numbers = Dictionary<Int, String>( )

辞書とエントリ

- 辞書[key]…keyの値を持つvalueを返す（オプショナル型）
- 辞書[key] = value…keyとvalueの組み合わせを登録
- 辞書.randomElement()? … 適当なエントリをタプルとして返す
- 辞書.removeValue(forKey: key)…指定されたキーのエントリを削除する
- 辞書.removeAll( )…全キー（エントリ）をクリアする
- 辞書.updateValue(新しい値, forKey: key) …指定されたキーの値を新しい値で置き換え、古い値を返す（オプショナル型なので、もし、古い値がなければnilが返される）
    - 例：let oldvalue = flowers.updateValue("ゆり", forKey: "lily")!

辞書型の属性

- 辞書.count … 辞書の中のエントリの個数を返す
- 辞書.isEmpty … 辞書が空かどうかを論理値で返す
- 辞書.contains(key) …keyが辞書にあるかかどうか論理値で返す
- 辞書.keys…辞書にあるすべてのキーを配列として返す
  - 例：**for** ename **in** flowers.keys.sorted() { print(flowers[ename]!) }
- 辞書.values…辞書にあるすべての値を配列として返す
- for (変数, 変数) in 辞書 { }…辞書の各エントリが、変数に代入される
  - 例：**for** (ename, jname) **in** flowers {

```
        print( "\(ename) is translated to \(jname)." )
    }
```

辞書のマージ

- 辞書.merge(他の辞書)
- 同じキーがある場合
  - var dictionary = ["a": 1, "b": 2]
  - // キー"a"の元の値を保持する
  - dictionary.merge(["a": 3, "c": 4]) { (current, _) in current }
  - // ["b": 2, "a": 1, "c": 4]
  - // キー"a"の新しい値で置き換える
  - dictionary.merge(["a": 5, "d": 6]) { (_, new) in new }
  - // ["b": 2, "a": 5, "c": 4, "d": 6]

配列(Array), 集合(Set), 辞書(Dictionary) , 範囲型(Range)に共通の関数

- enumerated()
- zip()
- filter()
- map()
- reduce()
- forEach()
- repeatElement()
- sequence()
- lazy()

ファイルからの読込み・ファイルへの書込み

- import Foundationが必要
- FileHandleオブジェクトを作成する
  - let f = FileHandle(forReadingAtPath: ファイル名)! …読込み
  - let f = FileHandle(forWritingAtPath: ファイル名)! …新規作成・上書き
  - let f = FileHandle(forUpdatingAtPath: ファイル名)! …更新用に読み書き
- let data:Data = f.read(upToCount: サイズ)! …指定された数読み込む
- let data:Data = f.readToEnd( )! …最後までの全体を読み込む
- f.close( )…アクセス終了
- Dataクラスを文字列型に変換するのには、下記の関数を使う（.utf8でもOK）
  - String(data: Data型の変数, encoding: String.Encoding.utf8)!

Webからの読込み

- **import** Foundationが必要
- URLオブジェクトを作成する
  - URL(string: URLの文字列)!
- Stringクラスのコンストラクタで、読込む
  - **try** String(contentsOf: urlの変数)
  - **try** String(contentsOf: urlの変数, encoding: エンコード)
  - エンコードは、String.Encodingクラスに一覧がある
- あとは、文字列として使える
  - 例：**let** pyurl = URL(string: "https://docs.python.org/3/")!
  - **let** ss = **try** String(contentsOf: pyurl)

ファイルのseekと書込み

- ファイル内の任意の場所に書込み（読込み）位置を移動する
  - f.seek(toOffset: バイトのオフセット) … 指定位置までポインタを移動
  - f.seekToEnd( ) … ファイルの最後までポインタを移動
  - f.offset( ) … 現在のポイントを返すUInt64

- f.write(データ)
  - ファイルの現在の位置に、データを書き込む
- 文字列をData型の変数に変換するには、下記の関数を使う（.utf8でもOK）
  - 文字列.data(using: String.Encoding.utf8)!
- f.truncate(atOffset: バイトのオフセット)
  - ファイルの内容を指定されたオフセットまでとする