Ubuntu Linux 上における Swift 言語の検証
swift.org - A Swift Tour を消化する

2019 年 5 月 14 日 (火)

abstract 要旨

Swift 言語の実行環境を Ubuntu Linux OS 上にダウンロードして実行した。swift.org の A Swift Tour という長い 1 ページのチュートリアルに取り組んだ。macOS や iOS に依存する部分がなく、言語仕様を学ぶ上で支障がなかった。

introduction 序論

Swift というプログラミング言語がある。各バージョンの登場時期は表の通りである。iOS 用プログラムを macOS 上で開発する際の実装にこの言語を用いることが念頭にある。しかしながらここでは、Ubuntu OS 上で Swift を用いることについてのみ考える。

Swift は Objective-C より入りやすいらしいが、実際には多機能なようだ。Ubuntu で Swift を使った場合には iOS や macOS のランタイムは使えないらしいが、言語仕様自体が大きいために、言語仕様を学ぶためには、Ubuntu で Swift を動かす意義もあるかもしれない。

date	version
2014-09-09	Swift 1.0
2014-10-22	Swift 1.1
2015-04-08	Swift 1.2
2015-09-21	Swift 2.0
2016-09-13	Swift 3.0
2017-09-19	Swift 4.0
2018-03-29	Swift 4.1
2018-09-17	Swift 4.2
2019-03-25	Swift 5.0

表 1　Swift language version history

sudo snap install swift

というパッケージがあるが現在は動作しないとのこと。
https://askubuntu.com/questions/1142200/how-can-i-install-swift-on-ubuntu-19-04/1142272

> Ubuntu has a swift snap package that is buggy and cannot be run at all. What "cannot be run at all" means is that not only does the swift snap package not run at all, but it can't be hacked to run at all without rebuilding the swift snap package. Hopefully this bug will be fixed soon, so that swift can be installed the nice way with sudo snap install swift
>
> Ubuntu に swift の snap パッケージがあるが、バグがあり、まったく実行できない。ここで「まったく実行できない」とはつまり、その swift の snap パッケージがまったく動かないのみならず、swift の snap パッケージをビルドしなおすのでなければ、実行するように調整することもできないという意味だ。望むらくは、バグが近く修正され、sudo snap install swift としてインストールできることを。

なおアンインストールは sudo snap remove swift としてできる。

installation

swift.org から「swift-5.0.1-RELEASE-ubuntu18.04.tar.gz」というファイルをダウンロードした。これを展開してそのなかの usr/bin/にある実行ファイルらを使えばよいらしい。

次のようにしてパスを通せる。

```
export PATH="~/swift-5.0.1-RELEASE-ubuntu18.04/usr/bin:$PATH"
```

hello, world

```
// $ swift helloworld.swift などとして実行できる。
print("hello, world")
print("こんにちは世界")
```

A Swift Tour

という解説が swift.org にある。

implementation

```
// A Swift Tour
// https://docs.swift.org/swift-book/GuidedTour/GuidedTour.html

// print()
print("Hello, world!")
```

```
// var, let; 変数と定数
var myVariable = 42
print(myVariable)
myVariable = 50      // 変数を更新する
let myConstant = 42  // 定数を定義する
print(myVariable, myConstant)

// type declaration
let implicitInteger = 70
let implicitDouble = 70.0
let explicitDouble: Double = 70  // 型を明示的に宣言する
print(implicitInteger, implicitDouble, explicitDouble)

// Experiment
// Create a constant with an explicit type of Float and a value of 4.
let explicitFloat: Float = 4  // 4 自体の型は Int だろう
print(explicitFloat)          // -> 4.0

// explicit type conversion
let label = "The width is "
let width = 94
let widthLabel = label + String(width)
print(label, width, widthLabel)

// Experiment
// Try removing the conversion to String from the last line. What error do
// you get?
// let widthLabel2 = label + width
// error: binary operator '+' cannot be applied to operands of type 'String'
//        and 'Int'
// note: overloads for '+' exist with these partially matching parameter
//       lists: (Int, Int), (String, String)

let apples = 3
let oranges = 5
let appleSummary = "I have \(apples) apples."
let fruitSummary = "I have \(apples + oranges) pieces of fruit."
print(apples, oranges, appleSummary, fruitSummary)
```

```
// Experiment
// Use \() to include a floating-point calculation in a string and to
// include someone's name in a greeting.
let someonesName = "John Appleseed"
print("\(someonesName) calculated \(1.2 + 3.4).")

let quotation = """
    I said "I have \(apples) apples."
    And then I said "I have \(apples + oranges) pieces of fruit."
    """
print(quotation)

var shoppingList = ["catfish", "water", "tulips"]  // 配列
print(shoppingList)
shoppingList[1] = "bottle of water"                // 配列を更新する
print(shoppingList)
var occupations = [                                // 辞書
    "Malcolm": "Captain",
    "Kaylee": "Mechanic",
]
print(occupations)
occupations["Jayne"] = "Public Relations"          // 辞書を更新する
print(occupations)

shoppingList.append("blue paint")
print(shoppingList)

let emptyArray = [String]()
let emptyDictionary = [String: Float]()
print(emptyArray, emptyDictionary)

shoppingList = []  // すでに使用され型が定まっているので型を明示しなくていい
occupations = [:]  // すでに使用され型が定まっているので型を明示しなくていい
print(shoppingList, occupations)

// control flow
let individualScores = [75, 43, 103, 87, 12]
var teamScore = 0
for score in individualScores {
    if score > 50 {
```

```
        teamScore += 3
    } else {
        teamScore += 1
    }
}
print(teamScore)  // 3 + 1 + 3 + 3 + 1 = 3*3 + 2*1 = 11

// optional value
var optionalString: String? = "Hello"
print(optionalString == nil)     // -> false
var optionalName: String? = "John Appleseed"
var greeting = "Hello!"
if let name = optionalName {     // optionalName が nil でなければ実行される
    greeting = "Hello, \(name)"  // 実行される
}
print(greeting)

// Experiment
// Change optionalName to nil. What greeting do you get? Add an else clause
// that sets a different greeting if optionalName is nil.
optionalName = nil
greeting = "Hello!"
if let name = optionalName {     // optionalName が nil でなければ実行される
    greeting = "Hello, \(name)"  // 実行されない
}
print(greeting)
if let name = optionalName {     // optionalName が nil でなければ実行される
    greeting = "Hello, \(name)"  // 実行されない
} else {
    greeting = "Hello!!"         // else 節が実行される
}
print(greeting)

// default value
let nickName: String? = nil
let fullName: String = "John Appleseed"
// ニックネームがあればそれで、なければフルネームで挨拶する。
let informalGreeting = "Hi \(nickName ?? fullName)"
print(informalGreeting)
```

```
let vegetable = "red pepper"
switch vegetable {
case "celery":  // セロリ
    print("Add some raisins and make ants on a log.")
    print("いくらかのレーズンを加えて丸太と蟻のようにしよう。")
case "cucumber", "watercress":  // キュウリとクレソン
    print("That would make a good tea sandwich.")
    print("ティータイムのサンドイッチを作ることができる。")
case let x where x.hasSuffix("pepper"):  // 接尾辞が pepper だ。
    print("Is is a spicy \(x)?")
default:
    print("Everything tastes good in soup.")
}

// Experiment
// Try removing the default case. What error do you get?
// error: switch must be exhaustive
// note: do you want to add a default clause?

let interestingNumbers = [
    "Prime":     [2,  3,  5,  7, 11, 13],  // 素数
    "Fibonacci": [1,  1,  2,  3,  5,  8],  // フィボナッチ数列
    "Square":    [1,  4,  9, 16, 25],      // 平方数
]
var largest = 0                              // 最大値
var largestKind = ""
for (kind, numbers) in interestingNumbers {  // 各数列について
    for number in numbers {                  // 各数について
        if number > largest {
            largest = number
            largestKind = kind
        }
    }
}
print(largest, largestKind)

// Experiment
// Add another variable to keep track of which kind of number was the
// largest, as well as what that largest number was.
```

```
var n = 2
while n < 100 {
    n *= 2
}
print(n)  // 100 以上で最小の 2 の倍数である 128 を出力する。

var m = 2
repeat {
    m *= 2
} while m < 100
print(m)  // この場合は上と同じで結果は 128 だ。

var total = 0
for i in 0..<4 {  // i     == 0, 1, 2, 3
    total += i    // total == 0 + 1 + 2 + 3
}
print(total)      // -> 6

// functions and closures
// 関数定義。関数 greet を定義する。
// greet() は String 型の person と String 型の day を取り String を返す。
func greet(person: String, day: String) -> String {
    return "Hello \(person), today is \(day)."  // 文字列を返す。
}
print(greet(person: "Bob", day: "Tuesday"))     // 関数呼び出し

// Experiment
// Remove the day parameter. Add a parameter to include today's lunch
// special in the greeting.
func greet(person: String, lunch: String) -> String {
    return "Hello \(person), today's lunch special is \(lunch)."
}
print(greet(person: "Appleseed", lunch: "TKG (tamago kake gohan)"))

// パラメータ person の引数名はなしとし、day の引数名は on とする。
func greet(_ person: String, on day: String) -> String {
    return "Hello \(person), today is \(day)."
}
print(greet("John", on: "Wednesday"))
```

```
// 複合的な値はタプルで返すことができる。
func calculateStatistics(scores: [Int]) -> (min: Int, max: Int, sum: Int) {
    var min = scores[0]  // 初期化
    var max = scores[0]
    var sum = 0

    for score in scores {  // 各スコアについて
        if score > max {   // 新しい最大値を見つけた
            max = score    // 更新
        } else if score < min {  // 常に max>=min だから同時には成り立たない
            min = score
        }
        sum += score       // 総和
    }

    return (min, max, sum)  // タプルを返す
}
let statistics = calculateStatistics(scores: [5, 3, 100, 3, 9])
print(statistics)
print(statistics.sum, statistics.2)  // -> 120 120

func returnFifteen() -> Int {
    var y = 10
    func add() {  // ネストした関数を定義する。
        y += 5    // 外側のスコープの変数 y にアクセス（変更）できる。
    }
    add()         // ネストした関数を呼び出す。
    return y
}
print(returnFifteen())

// 引数 +1 を返す関数を返す関数。
func makeIncrementer() -> ((Int) -> Int) {
    // 引数 +1 を返す関数。
    func addOne(number: Int) -> Int {
        return 1 + number
    }
    return addOne
}
var increment = makeIncrementer()  // 関数
```

```
print(increment(7))                // -> 8

// 関数を取る関数。
// list の要素で condition なものがあれば true を返す。
func hasAnyMatches(list: [Int], condition: (Int) -> Bool) -> Bool {
    for item in list {         // リストの各要素について
        if condition(item) {   // condition なものが見つかった。
            return true        // さらに探さない
        }
    }
    return false               // 1 つも condition ではなかった
}
func lessThanTen(number: Int) -> Bool {  // number は 10 未満か
    return number < 10
}
var numbers = [20, 19, 7, 12]  // 10 未満な数 7 を持つリスト
print(hasAnyMatches(list: numbers, condition: lessThanTen))  // -> true

// 無名関数を定義する。
print(numbers.map({ (number: Int) -> Int in
    let result = 3 * number
    return result
}))  // -> [60, 57, 21, 36]

// Experiment
// Rewrite the closure to return zero for all odd numbers.
print(numbers.map({ (number: Int) -> Int in
    if number % 2 == 1 { return 0 }
    return 3 * number
}))  // -> [60, 0, 0, 36]

// 推測できる型宣言は省略できる。唯一の文なら return は省略できる。
let mappedNumbers = numbers.map({ number in 3 * number })
print(mappedNumbers)  // -> [60, 57, 21, 36]

let sortedNumbers = numbers.sorted { $0 > $1 }
print(sortedNumbers)  // -> [20, 19, 12, 7]

class Shape {              // クラス定義
    var numberOfSides = 0  // プロパティ宣言
```

```
    let s = "shape"
    func simpleDescription() -> String {  // メソッド宣言
        return "A \(s) with \(numberOfSides) sides."
    }
    func anotherMethod(_ arg: String) {
        print(arg)
    }
}
// Experiment
// Add a constant property with let, and add another method that takes
// an argument.
var shape = Shape()      // インスタンス生成
shape.numberOfSides = 7  // プロパティへのアクセス
var shapeDescription = shape.simpleDescription()  // 関数へのアクセス
shape.anotherMethod(shapeDescription)

// イニシャライザのあるクラス。
class NamedShape {
    var numberOfSides: Int = 0
    var name: String
    init(name: String) {  // イニシャライザ
        self.name = name  // self. によるプロパティアクセス
    }
    func simpleDescription() -> String {
        return "A shape \(name) with \(numberOfSides) sides."
    }
}
let shape2 = NamedShape(name: "Pentagon")
shape2.numberOfSides = 5
print(shape2.simpleDescription())

// サブクラス
class Square: NamedShape {      // 継承
    var sideLength: Double      // このサブクラスで追加されたプロパティ
    init(sideLength: Double, name: String) {
        self.sideLength = sideLength
        super.init(name: name)  // スーパークラスのイニシャライザ呼び出し
        numberOfSides = 4
    }
    func area() -> Double {  // 面積を返す
```

```
        return sideLength * sideLength
    }
    override func simpleDescription() -> String {  // メソッドのオーバーライド
        return "A square with sides of length \(sideLength)."
    }
}
let test = Square(sideLength: 5.2, name: "my test square")
print(test.area(), test.simpleDescription())


// Experiment
// Make another subclass of NamedShape called Circle that takes a radius
// and a name as arguments to its initializer. Implement an area() and a
// simpleDescription() method on the Circle class.
class Circle: NamedShape {
    var radius: Double  // 半径
    init(radius: Double, name: String) {
        self.radius = radius
        super.init(name: name)
        self.numberOfSides = 1
    }
    func area() -> Double {
        return 3.14 * radius * radius  // pi*r^2
    }
    override func simpleDescription() -> String {
        return "A circle \(name) with area of \(area())."
    }
}
print(Circle(radius: 2, name: "Circle2").simpleDescription())

// ゲッタとセッタのあるプロパティのあるクラス
class EquilateralTriangle: NamedShape {  // 正三角形
    var sideLength: Double = 0.0         // 1 辺の長さ
    init(sideLength: Double, name: String) {
        self.sideLength = sideLength     // サブクラスのプロパティの値を定める
        super.init(name: name)           // スーパークラスのイニシャライザ
        numberOfSides = 3                // スーパークラスのプロパティの値を定める
    }
    var perimeter: Double {              // 周長
        get {
```

```
            return 3.0 * sideLength
        }
        set {
            sideLength = newValue / 3.0  // 暗黙の変数名 newValue を使う
        }
    }
    override func simpleDescription() -> String {
        return "An equilateral triangle with sides of length \(sideLength)."
    }
}
var triangle = EquilateralTriangle(sideLength: 3.1, name: "a triangle")
print(triangle.perimeter)  // -> 9.3
triangle.perimeter = 9.9
print(triangle.sideLength) // -> 3.3
print(triangle.simpleDescription())

class TriangleAndSquare {  // 三角形と四角形をプロパティに持つクラス
    var triangle: EquilateralTriangle {
        willSet {  // triangle に newValue が代入される直前に square を同期する。
            square.sideLength = newValue.sideLength
        }
    }
    var square: Square {
        willSet {
            triangle.sideLength = newValue.sideLength
        }
    }
    init(size: Double, name: String) {
        // プロパティを初期化する。init() からは willSet は発動しない。
        square = Square(sideLength: size, name: name)
        triangle = EquilateralTriangle(sideLength: size, name: name)
    }
}
var triangleAndSquare = TriangleAndSquare(size: 10, name: "another test shape")
print(triangleAndSquare.square.sideLength)    // -> 10
print(triangleAndSquare.triangle.sideLength)  // -> 10
triangleAndSquare.square = Square(sideLength: 50, name: "larger square")
print(triangleAndSquare.triangle.sideLength)  // -> 50 （同期されている）

// ? はメソッド、プロパティ、添字の前で使える。nil なら後部は無視される。
```

```
// nil でないなら、unwrap された値として操作され wrap されて終わる。
var optionalSquare: Square? = Square(sideLength: 2.5, name: "optional square")
var sideLength = optionalSquare?.sideLength
print(sideLength ?? "NIL")
optionalSquare = nil  // 変数 optionalSquare はオプショナル型なので nil を表せる
sideLength = optionalSquare?.sideLength  // nil なので sideLength() されない
print(sideLength ?? "NIL")               // nil なのでデフォルト値を出力する

// enumeration を作成し、enumeration case らを定義する。
enum Rank: Int {
    case ace = 1  // デフォルトは 0
    case two, three, four, five, six, seven, eight, nine, ten
    case jack, queen, king
    func simpleDescription() -> String {  // enumeration はメソッドを持てる
        switch self {
        case .ace:
            return "ace"
        case .jack:
            return "jack"
        case .queen:
            return "queen"
        case .king:
            return "king"
        default:
            return String(self.rawValue)  // 暗黙に存在する rawValue プロパティ
        }
    }
}
let ace = Rank.ace
let aceRawValue = ace.rawValue
print(aceRawValue)

// Experiment
// Write a function that compares two Rank values by comparing their raw values.
func isGreaterRank(x: Rank, y: Rank) -> Bool {
    return x.rawValue > y.rawValue
}
print(isGreaterRank(x: Rank.king, y: Rank.ace),    // -> true
      isGreaterRank(x: Rank.two,  y: Rank.queen))  // -> false
```

```
if let convertedRank = Rank(rawValue: 3) {
    let threeDescription = convertedRank.simpleDescription()
    print(threeDescription)  // -> 3
}


enum Suit {  // 具体的な値に紐づけずに enumeration を定義していい
    case spades, hearts, diamonds, clubs
    func simpleDescription() -> String {
        switch self {
        case .spades:
            return "spades"
        case .hearts:
            return "hearts"
        case .diamonds:
            return "diamonds"
        case .clubs:
            return "clubs"
        }
    }
    func color() -> String {
        switch self {
        case .spades, .clubs:
            return "black"
        case .hearts, .diamonds:
            return "red"
        }
    }
}
let hearts = Suit.hearts
let heartsDescription = hearts.simpleDescription()
print(heartsDescription)  // -> "hearts"
// Experiment
// Add a color() method to Suit that returns “black” for spades and clubs,
// and returns “red” for hearts and diamonds.
print(hearts.color())

// enumeration ケースに属する値がある enumeration。
enum ServerResponse {
    case result(String, String)  // 属する値はインスタンス化の際に決まる
```

```
    case failure(String)          // 属する値はケースのプロパティのようなものだ
    case asInt(Int, Int)
}
// サーバに日の出と日没の時刻を問い合わせる。チーズが不足する場合にはエラーが返る。
for x in [ServerResponse.result("6:00 am", "8:09 pm"),
          ServerResponse.failure("Out of cheese"),
          ServerResponse.asInt(6, 8)] {
    switch x {
    case let .result(sunrise, sunset):
        print("Sunrise is at \(sunrise) and sunset is at \(sunset).")
    case let .failure(message):
        print("Failure... \(message)")
    case let .asInt(x, y):
        print("sunrise=\(x), sunset=\(y), sum=\(x+y).")
    }
}
// Experiment
// Add a third case to ServerResponse and to the switch.

// 構造体は pass by value である点でクラスと異なる。
struct Card {
    var rank: Rank
    var suit: Suit
    func simpleDescription() -> String {
        return "The \(rank.simpleDescription()) of \(suit.simpleDescription())"
    }
}
let threeOfSpades = Card(rank: .three, suit: .spades)
let threeOfSpadesDescription = threeOfSpades.simpleDescription()
print(threeOfSpadesDescription)  // -> The 3 of spades

// Experiment
// Write a function that returns an array containing a full deck of cards,
// with one card of each combination of rank and suit.
func getFullDeck() -> [Card] {
    var cards: [Card] = []
    for suit in [Suit.spades, Suit.hearts, Suit.diamonds, Suit.clubs] {
        for n_rank in 1...13 {  // 理想的な解決ではないだろう
            let rank = Rank(rawValue: n_rank) ?? Rank.ace
            cards.append(Card(rank: rank, suit: suit))
```

```
        }
    }
    return cards
}
print(getFullDeck().map { $0.simpleDescription() })

// プロトコルの宣言。
protocol ExampleProtocol {
    var simpleDescription: String { get }
    mutating func adjust()  // struct や enum の変更には mutating 修飾子が必要だ。
    var anotherProperty: Int { get }
}

class SimpleClass: ExampleProtocol {
    var simpleDescription: String = "A very simple class."
    var anotherProperty: Int = 69105
    func adjust() {
        simpleDescription += "  Now 100% adjusted."
    }
}
var a = SimpleClass()
a.adjust()
let aDescription = a.simpleDescription
print(aDescription)

struct SimpleStructure: ExampleProtocol {
    var simpleDescription: String = "A simple structure"
    var anotherProperty: Int = 69105
    mutating func adjust() {
        simpleDescription += " (adjusted)"
    }
}
var b = SimpleStructure()
b.adjust()
let bDescription = b.simpleDescription
print(bDescription)
// Experiment
// Add another requirement to ExampleProtocol. What changes do you need to
// make to SimpleClass and SimpleStructure so that they still conform to
// the protocol?
```

```
// extension を用いる。既存の型に機能やプロトコルを追加できる。
extension Int: ExampleProtocol {
    var simpleDescription: String {
        return "The number \(self)"
    }
    var anotherProperty: Int {
        return self
    }
    mutating func adjust() {
        self += 42
    }
}
print(7.simpleDescription)

// Experiment
// Write an extension for the Double type that adds an absoluteValue property.
extension Double {
    var absoluteValue: Double {
        if self >= 0 { return self }
        return -self
    }
}
print(7.0.absoluteValue, (-7.0).absoluteValue)  // -> 7 7

// プロトコルは型として使える。ただしそれを通しては、プロトコルのメンバしか使えない。
let protocolValue: ExampleProtocol = a
print(protocolValue.simpleDescription)

// Error プロトコルを受容する任意の型によってエラーを表せる。
enum PrinterError: Error {
    case outOfPaper  // 紙がない
    case noToner     // トナーがない
    case onFire      // 燃えている
}

// エラーを投げうる関数を定義する。
func send(job: Int, toPrinter printerName: String) throws -> String {
    if printerName == "Never Has Toner" {
        throw PrinterError.noToner  // エラーを投げる
```

```
    }
    return "Job sent"
}
do {
    // 畢昇 (Bi Sheng) は活版印刷術の発明者である。
    let printerResponse = try send(job: 1040, toPrinter: "Bi Sheng")
    print(printerResponse)
    print(try send(job: 1040, toPrinter: "Never Has Toner"))
} catch {
    print(error)  // 暗黙の変数名 error
}

// Experiment
// Change the printer name to "Never Has Toner", so that the
// send(job:toPrinter:) function throws an error.
// https://stackoverflow.com/questions/37472434/catching-a-default-error-in-swift

do {
    let printerResponse = try send(job: 1440, toPrinter: "Gutenberg")
    print(printerResponse)
    // throw PrinterError.onFire
    // throw PrinterError.outOfPaper
    // enum Error2: Error { case x }; throw Error2.x
} catch PrinterError.onFire {
    print("I'll just put this over here, with the rest of the fire.")
} catch let printerError as PrinterError {
    print("Printer error: \(printerError).")
} catch {
    print(error)
}

// try?を使えばエラーは nil になるので do ... catch する必要がない。
for x in [try? send(job: 1884, toPrinter: "Mergenthaler"),
          try? send(job: 1885, toPrinter: "Never Has Toner")] {
    print(x ?? "NIL")
}

var fridgeIsOpen = false
let fridgeContent = ["milk", "eggs", "leftovers"]
func fridgeContains(_ food: String) -> Bool {
```

```
    fridgeIsOpen = true
    defer {  // defer（延期する）により実行を保証する。
        fridgeIsOpen = false
    }
    let result = fridgeContent.contains(food)
    return result
}
print(fridgeContains("banana"))  // -> false
print(fridgeIsOpen)              // -> false

// generics; 型の総称を<Item>として用いている。
func makeArray<Item>(repeating item: Item, numberOfTimes: Int) -> [Item] {
    var result = [Item]()
    for _ in 0..<numberOfTimes {
        result.append(item)
    }
    return result
}
print(makeArray(repeating: "knock", numberOfTimes: 4))

enum OptionalValue<Wrapped> {
    case none
    case some(Wrapped)
}
var possibleInteger: OptionalValue<Int> = .none
print(possibleInteger)
possibleInteger = .some(100)
print(possibleInteger)

// O(n^2). where を用いて総称型を制約する。
// 型 T と U はともに Sequence でその要素の型が等しく、要素の等しさが判定できる。
func anyCommonElements<T: Sequence, U: Sequence>(_ lhs: T, _ rhs: U) -> Bool
        where T.Element: Equatable, T.Element == U.Element {
    for lhsItem in lhs {
        for rhsItem in rhs {
            if lhsItem == rhsItem {
                return true  // 同じ要素が見つかった
            }
        }
    }
```

```
    return false              // 同じ要素が見つからなかった
}
print(anyCommonElements([1, 2, 3], [3]))  // -> true

// Experiment
// Modify the anyCommonElements(_:_:) function to make a function that
// returns an array of the elements that any two sequences have in common.
func anyCommonElements2<T: Sequence, U: Sequence>(_ lhs: T, _ rhs: U) -> [T.Element]
        where T.Element: Equatable, T.Element == U.Element {
    var items = [T.Element]()
    for lhsItem in lhs {
        for rhsItem in rhs {
            if lhsItem == rhsItem {
                items.append(lhsItem)
            }
        }
    }
    return items
}
print(anyCommonElements2([1, 2, 3], [3]))  // -> [3]
```

If not let - in Swift

https://stackoverflow.com/questions/27412735/if-not-let-in-swift
if not let x = ... と書くことはできない。if x == nil と書けばいい。

conclusion 結論

Swift 言語の最も基本的な概観に触れることができた。言語仕様の学習であったために、Ubuntu OS 上で問題なく取り組めた。