Learning Swift Language on Ubuntu:
The Basics
Basic Operators
Strings and Characters
Collection Types
Control Flow

language guide - the basics

implementation

```
// language guide - the basics
// https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html

let maximumNumberOfLoginAttempts = 10  // 定数
var currentLoginAttempt = 0            // 変数
print(maximumNumberOfLoginAttempts, currentLoginAttempt)

var x = 0.0, y = 0.0, z = 0.0  // 複数の変数を 1 行で宣言する
print(x, y, z)

var welcomeMessage: String  // 型注釈を使う。代入前に使うとコンパイルエラー。
welcomeMessage = "Hello"
print(welcomeMessage)

var red, green, blue: Double  // 型注釈が行全体に効く

let π = 3.14159, 你好 = "你好世界",      = "dogcow"  // unicode
print("π, 你好,     ", π, 你好,     )

let `if` = 1  // キーワードをどうしても変数名にしたければ`で囲む
```

```
print(‘if‘)

var friendlyWelcome = "Hello!"
friendlyWelcome = "Bonjour!"
print(friendlyWelcome)

let languageName = "Swift"
print(languageName + "++")

print("your age: ", terminator: "")  // 改行以外で終わりたければ指定する
print(99)

// string interpolation 文字列補間
print("The current value of friendlyWelcome is \(friendlyWelcome)")

/* /* ネストした複数行コメント */ */

let cat = "  "; print(cat)  // セミコロンを使えば 1 行に複数の文を書ける
let uInt8Value: UInt8 = 255; print(uInt8Value)
print(UInt8.min, UInt8.max, type(of: UInt8.max))
print(Int.min, Int.max, type(of: Int.max))

// Unless you need to work with a specific size of integer, always use
// Int for integer values in your code.
// 特に必要がない限りはサイズを指定せず Int 型を用いるべきだ。

// Use UInt only when you specifically need an unsigned integer type with
// the same size as the platform's native word size. If this isn't the
// case, Int is preferred, even when the values to be stored are known to
// be nonnegative.
// 特に必要がない限りは正の数に限られる状況でも単に Int 型を用いるべきだ。

// Double has a precision of at least 15 decimal digits, whereas the
// precision of Float can be as little as 6 decimal digits.
// ...
// In situations where either type would be appropriate, Double is preferred.
// 特に必要がないなら Float より Double を使っておけばいい。

let meaningOfLife = 42, pi = 3.14159          // 型推論
print(type(of: meaningOfLife), type(of: pi))  // -> Int Double
```

```
print(17, 0b10001, 0o21, 0x11)  // 10 進数、2 進数、8 進数、16 進数で 17

print(1.25e2, 1.25e-2)
print(0xfp2, 0xfp-2)
print(12.1875, 1.21875e1, 0xc.3p0)
print(000123.456, 1_000_000, 1_000_000.000_000_1, 1_0000)

// 数の型変換
// var tooBig: UInt8 = 255; tooBig += 1  // 実行時エラー

let twoThousand: UInt16 = 2_000
let one: UInt8 = 1
let twoThousandAndOne = twoThousand + UInt16(one)  // 新インスタンス作成
print(twoThousandAndOne)

let three = 3
let pointOneFourOneFiveNine = 0.14159
let pi2 = Double(three) + pointOneFourOneFiveNine
print(pi2)
print(Int(pi2))

// 型の別名 (type aliase)
typealias AudioSample = UInt16
var maxAmplitudeFound = AudioSample.min
print(maxAmplitudeFound, type(of: maxAmplitudeFound))  // -> 0 UInt16

// Bool
let orangesAreOrange = true, turnipsAreDelicious = false  // カブ
print(orangesAreOrange, turnipsAreDelicious)
if turnipsAreDelicious {  // 条件部が Bool 型でなければコンパイルエラー
    print("Mmm, tasty turnips!")
} else {
    print("Eww, turnips are horrible.")
}

// タプル
let http404Error = (404, "Not Found")
print(http404Error, type(of: http404Error))
let (statusCode, statusMessage) = http404Error  // decompose
```

```
print(statusCode, statusMessage, http404Error.0)
let http200Status = (statusCode: 200, description: "OK")  // named tuple
print(http200Status.statusCode, http200Status.description)

// If your data structure is likely to persist beyond a temporary scope,
// model it as a class or structure, rather than as a tuple.
// 多少なり長く使う構造にタプルを汎用すべきではない。クラスなどが適している。

// optional
let possibleNumber = "123"
let convertedNumber = Int(possibleNumber)  // Int(String)は:Int?である。
print(convertedNumber as Any, type(of: convertedNumber))

var serverResponseCode: Int? = 404
print(serverResponseCode as Any)
serverResponseCode = nil
print(serverResponseCode as Any)

var surveyAnswer: String?; print(surveyAnswer as Any)  // 初期値nil

if convertedNumber != nil {
    print("convertedNumber contains some integer value.")
    // 文脈上optionalがnilでないと確信できるのでforced unwrappingする。
    print("convertedNumber has an integer value of \(convertedNumber!).")
}

if let actualNumber = Int(possibleNumber) {
    print("The string \"\(possibleNumber)\" has an integer value " +
          "of \(actualNumber)")
} else {
    print("The string \"\(possibleNumber)\" could not be converted " +
          "to an integer")
}

// 条件部を,で並べれば&&のような意味になる。
if let firstNumber = Int("4"), let secondNumber = Int("42"),
        firstNumber < secondNumber, secondNumber < 100 {
    print("\(firstNumber) < \(secondNumber) < 100")
}
```

```
// implicitly unwrapped optional というものがある。
let possibleString: String? = "An optional string."
let forcedString: String = possibleString!  // forced unwrapping
let assumedString: String! = "An implicitly unwrapped optional string."
let implicitString: String = assumedString
print(forcedString, implicitString)
print(type(of: possibleString), type(of: assumedString))  // String? String?
if assumedString != nil {
    print(assumedString!)
    print(assumedString as Any)
}
if let definiteString = assumedString {
    print(definiteString)
}

// Stopping execution as soon as an invalid state is detected also helps
// limit the damage caused by that invalid state.

let age = -3
// assert(age >= 0)
// assert(age >= 0, "A person's age can't be less than zero.")

if age > 10 {
    print("You can ride the roller-coaster or the ferris wheel.")
} else if age >= 0 {
    print("You can ride the ferris wheel.")
} else {
    // assertionFailure("A person's age can't be less than zero.")
}

// The difference between assertions and preconditions is in when they' re
// checked: Assertions are checked only in debug builds, but preconditions
// are checked in both debug and production builds.
// assert() はデバッグビルドでのみ有効だが precondition() はそうではない。
// ただし、precondition() も-Ounchecked としてオフにされうる。
// precondition(age >= 0)

// 決してオフにされないのは fatalError() だけだ。しかし assert() の機能はない。
// fatalError("Unimplemented")
```

implementation

```
// basic operators
// https://docs.swift.org/swift-book/LanguageGuide/BasicOperators.html

print(1..<10, 1...10, type(of: 1..<10), type(of: 1...10))
// print(! true)  // compile-time error
for x in [true, false] {
    print(x ? "T" : "F")  // C 同様に唯一の三項演算子
}

// assignment operator 代入演算子
let b = 10
var a = 5
a = b
print(a, b)  // -> 10 10

let (x, y) = (1, 2)  // decompose
print(x, y)  // -> 1 2

// arithmetic operators 算術演算子
print(1 + 2, 5 - 3, 2 * 3, 10.0 / 2.5)  // 3 2 6 4.0
print("hello, " + "world")

// remainder operator 剰余演算子
print(9 % 4)   // 1
print(-9 % 4)  // -1; C と同じで Python と異なる。数学的な modulo ではない。

// 単項プラス／マイナス演算子。空白をあけてはならない。
let three = 3, minusThree = -three, plusThree = -minusThree
print(plusThree)
let minusSix = -6, alsoMinusSix = +minusSix
print(alsoMinusSix)

// compound assignment operators 複合代入演算子
a = 1; a += 2; print(a)  // 3

// comparison operators 比較演算子
// 2!=1 とは書けない。2!が不正な forced unwrapping と見なされるため。
```

```
// true true true true true false
print(1==1, 2 != 1, 2>1, 1<2, 1>=1, 2<=1)

let name = "world"
if name == "world" {
    print("hello, world")  // 実行される
} else {
    print("I'm sorry \(name), but I don't recognize you")
}

print( (1, "zebra") < (2, "apple"),  // true
       (3, "apple") < (3, "bird"),   // true
       (4, "dog") == (4, "dog"))     // true

print( ("blue", -1) < ("purple", 1) )  // true
// print( ("blue", false) < ("purple", true) )  // compile-time error
print((1,1,1,1,1,1)==(1,1,1,1,1,1))  // true
// print((1,1,1,1,1,1,1)==(1,1,1,1,1,1,1))  // compile-time error
// ∵ Swift 標準ライブラリは要素数が 6 個以下のタプルにしか比較演算子を用意していない。
// 必要なら自ら実装すればよい。

// 三項条件演算子
let contentHeight = 40
let hasHeader = true
print(contentHeight + (hasHeader ? 50 : 20))  // 90

// nil-coalescing operator ??
let defaultColorName = "red"
var userDefinedColorName: String?
var colorNameToUse = userDefinedColorName ?? defaultColorName
print(colorNameToUse)  // red. nil だったので右側オペランドが評価された。

// range operators 範囲演算子
for index in 1...5 {
    print("\(index) times 5 is \(index * 5)")
}
// var lowerBound = 200, upperBound = 100
// for index in lowerBound...upperBound {}  // runtime error

// half-open range operator
```

```
let names = ["Anna", "Alex", "Brian", "Jack"]
let count = names.count
for i in 0..<count {  // [0, count)
    print("Person \(i + 1) is called \(names[i])")
}

for name in names[2...] { print(name) }
for name in names[...2] { print(name) }
for name in names[..<2] { print(name) }

for i in 1000... { print(i); if i==1003 {break} }
// false true true
print((...5).contains(7), (...5).contains(4), (...5).contains(-1))

let allowedEntry = false
if !allowedEntry {
    print("ACCESS DENIED")
}
```

implementation

```
// strings and characters
// https://docs.swift.org/swift-book/LanguageGuide/StringsAndCharacters.html

print(type(of: ""))  // -> String
print("\"\u{24} \u{2665} \u{1f496}\"")
print(#"Line 1\nLine 2"#)  // extended string delimiters
print(###"Line 1\###nLine 2"###)
// print(##"""
//    abc
//    def
//    """##)
for s in ["", String()] {
    print(s.isEmpty, s=="")
}

// Behind the scenes, Swift's compiler optimizes string usage so that
// actual copying takes place only when absolutely necessary.
// 文字列は値渡しだが、自動的な最適化により、不要なコピーは実際には行われない。
```

```
for character in "Dog!  " {
    print(character)
}

let exclamationMark: Character = "!"
print(exclamationMark, type(of: exclamationMark))

let catCharacters: [Character] = ["C", "a", "t", "!", "  "]
let catString = String(catCharacters)
print(catString)

let string1 = "hello"
let string2 = " there"
var welcome = string1 + string2
welcome.append(exclamationMark)
print(welcome)

// string interpolation
let multiplier = 3
let message = "\(multiplier) times 2.5 is \(Double(multiplier) * 2.5)"
print(message)

print("\u{1f425}")

// extended grapheme cluster 拡張書記素クラスタ

let eAcute: Character = "\u{e9}"
let combinedEAcute: Character = "\u{65}\u{301}"
print(eAcute, combinedEAcute)

let precomposed: Character = "\u{D55C}"                  //
let decomposed: Character = "\u{1112}\u{1161}\u{11AB}"  //   、   、
print(precomposed, decomposed)

let regionalIndicatorForUS: Character = "\u{1F1FA}\u{1F1F8}"
print(regionalIndicatorForUS)

let unusualMenagerie = "Koala   , Snail   , Penguin   , Dromedary   "
print("unusualMenagerie has \(unusualMenagerie.count) characters")  // 40
```

```
var greeting = "Guten Tag!"
print(greeting[greeting.startIndex])                          // G
print(greeting[greeting.index(before: greeting.endIndex)])    // !
print(greeting[greeting.index(after: greeting.startIndex)])   // u
var index = greeting.index(greeting.startIndex, offsetBy: 7)
print(greeting[index])                                        // a

for index in greeting.indices {
    print("\(greeting[index]) ", terminator: "")
}
print()

welcome = "hello"
welcome.insert("!", at: welcome.endIndex)
print(welcome)
welcome.insert(contentsOf: " there",
               at: welcome.index(before: welcome.endIndex))
print(welcome)

welcome.remove(at: welcome.index(before: welcome.endIndex))
print(welcome)
let range = welcome.index(welcome.endIndex, offsetBy: -6)..<welcome.endIndex
welcome.removeSubrange(range)
print(welcome)

// Substring type
greeting = "Hello, world!"
index = greeting.firstIndex(of: ",") ?? greeting.endIndex
let beginning = greeting[..<index]
print(beginning, type(of: beginning))
let newString = String(beginning)
print(newString, type(of: newString))

print("\u{E9}", "\u{65}\u{301}", "\u{E9}"=="\u{65}\u{301}")  // true

// String and character comparisons in Swift are not locale-sensitive.

print("abcd".hasPrefix("ab"), "abcd".hasSuffix("cd"))  // true true
```

```
let dogString = "Dog‼  "
print(dogString)
print(type(of: dogString.utf8),            // UTF8View
      type(of: dogString.utf16),           // UTF16View
      type(of: dogString.unicodeScalars))  // UnicodeScalarView
print(type(of: dogString.utf8[dogString.utf8.startIndex]))    // UInt8
print(type(of: dogString.utf16[dogString.utf16.startIndex]))  // UInt16
print(type(of: dogString.unicodeScalars[
        dogString.unicodeScalars.startIndex]))  // Scalar
```

implementation

```
// collection types
// https://docs.swift.org/swift-book/LanguageGuide/CollectionTypes.html

// 基本的なコレクション型として、配列、セット、辞書がある。
// あるコレクションオブジェクトのすべての要素の型は同じである。
// 変数で参照したコレクションは可変であり、定数で参照したコレクションは不変である。

var someInts = [Int]()
var someInts1 = Array<Int>()
var someInts2: [Int] = []
print(someInts, someInts1, someInts2)

var threeDoubles = Array(repeating: 0.0, count: 3)
print(threeDoubles)
var anotherThreeDoubles = Array(repeating: 2.5, count: 3)
var sixDoubles = threeDoubles + anotherThreeDoubles
print(sixDoubles)

var shoppingList = ["Eggs", "Milk"]
print(shoppingList, type(of: shoppingList))  // Array<String>
var test1 = [1, 2.0]
print(test1, type(of: test1))  // Array<Double>

shoppingList.append("Flour")
shoppingList += ["Baking Powder"]
shoppingList += ["Chocolate Spread", "Cheese", "Butter"]
print(shoppingList)
```

```
shoppingList[0] = "Six eggs"
print(shoppingList)
shoppingList[4...6] = ["Bananas", "Apples"]  // change a range
print(shoppingList)
shoppingList.insert("Maple Syrup", at: 0)
print(shoppingList)
let mapleSyrup = shoppingList.remove(at: 0)
print(mapleSyrup, shoppingList)
let apples = shoppingList.removeLast()  // 特に最後の要素を削除する
print(apples, shoppingList)

for (index, var value) in shoppingList.enumerated() {
    value += "!"
    print("Item \(index + 1): \(value)")
}

// Set 集合; Set を表すリテラルはない。
var letters = Set<Character>()
letters.insert("a")
print(letters)
letters = []
print(letters)

// var favoriteGenres: Set<String> = ["Rock", "Classical", "Hip hop"]
var favoriteGenres: Set = ["Rock", "Classical", "Hip hop"]
print(favoriteGenres)
print("I have \(favoriteGenres.count) favorite music genres.")
if favoriteGenres.isEmpty {
    print("As far as music goes, I'm not picky.")
} else {
    print("I have particular music preferences.")
}
favoriteGenres.insert("Jazz")
if let removedGenre = favoriteGenres.remove("Rock") {
    print("\(removedGenre)? I'm over it.")
} else {
    print("I never much cared for that.")
}
if favoriteGenres.contains("Funk") {
    print("I get up on the good foot.")
```

```
} else {
    print("It's too funky in here.")
}
for genre in favoriteGenres {
    print("\(genre)")
}

let oddDigits              : Set = [1, 3, 5, 7, 9],
    evenDigits             : Set = [0, 2, 4, 6, 8],
    singleDigitPrimeNumbers: Set = [2, 3, 5, 7]
// [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] [] [1, 9] [1, 2, 9]
print(oddDigits.union(evenDigits).sorted(),
      oddDigits.intersection(evenDigits).sorted(),
      oddDigits.subtracting(singleDigitPrimeNumbers).sorted(),
      oddDigits.symmetricDifference(singleDigitPrimeNumbers).sorted())

let houseAnimals: Set = ["  ", "  "],
    farmAnimals : Set = ["  ", "  ", "  ", "  ", "  "],
    cityAnimals : Set = ["  ", "  "]
// true true true
print(houseAnimals.isSubset(of: farmAnimals),
      farmAnimals.isSuperset(of: houseAnimals),
      farmAnimals.isDisjoint(with: cityAnimals))

// Dictionary 辞書
var namesOfIntegers = [Int: String]()
// var namesOfIntegers: [Int: String] = [:]
// var namesOfIntegers: Dictionary<Int, String> = [:]
print(namesOfIntegers)
namesOfIntegers[16] = "sixteen"
print(namesOfIntegers)
namesOfIntegers = [:]
print(namesOfIntegers)

// International Air Transport Association codes
var airports = ["YYZ": "Toronto Pearson",
                "DUB": "Dublin"]
print(airports)
airports["LHR"] = "London"
print(airports)
```

```
airports["LHR"] = "London Heathrow"
print(airports)

// .updateValue() は添字による更新と同じだが古い値があれば返す。
if let oldValue = airports.updateValue("Dublin Airport", forKey: "DUB") {
    print("The old value for DUB was \(oldValue).")
}

// 辞書要素へのアクセスは Optional オブジェクトを返す。
if let airportName = airports["DUB"] {
    print("The name of the airport is \(airportName).")
} else {
    print("That airport is not in the airports dictionary.")
}

// 辞書の要素を削除するには nil を代入すればいい。
airports["APL"] = "Apple International"
print(airports)
airports["APL"] = nil
print(airports)

// .removeValue() で削除するとキーに対応する値が返される。
if let removedValue = airports.removeValue(forKey: "DUB") {
    print("The removed airport's name is \(removedValue).")
} else {
    print("The airports dictionary does not contain a value for DUB.")
}

// iterationg over a dictionary
for (airportCode, airportName) in airports {  // key, value を走査する
    print("\(airportCode): \(airportName)")
}
for airportCode in airports.keys {            // key を走査する
    print("Airport code: \(airportCode)")
}
for airportName in airports.values {          // value を走査する
    print("Airport name: \(airportName)")
}

print(type(of: airports.keys), type(of: airports.values))  // Keys Values
```

```
let airportCodes = [String](airports.keys),  // 配列にする
    airportNames = [String](airports.values)
print(airportCodes, airportNames)
```

implementation

```
// control flow 制御構造
// https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html

// for-in loop
let names = ["Anna", "Alex", "Brian", "Jack"]
for name in names {
    print("Hello, \(name)!")
}

// 辞書の key-value ペアらを for-in ループで decompose する。
let numberOfLegs = ["spider": 8, "ant": 6, "cat": 4]
for (animalName, legCount) in numberOfLegs {
    print("\(animalName)s have \(legCount) legs")
}

// Range (ClosedRange) オブジェクトを用いた for-in ループ。
for index in 1...5 {  // closed range operator
    print("\(index) times 5 is \(index * 5)")
}

let base=3, power=10
var answer=1
for _ in 1...power {
    // print(_)  // compile-time error; 単なる慣習ではない
    answer *= base
}
print("\(base)^\(power) == \(answer)")


print(type(of: stride(from: 0, to:      10, by: 2)))  // StrideTo<Int>
print(type(of: stride(from: 0, through: 10, by: 2)))  // StrideThrough<Int>
let minutes = 60
```

```
let minuteInterval = 5
for tickMark in stride(from: 0, to: minutes, by: minuteInterval) {
    print("\(tickMark), ", terminator:"")
}; print()
let hours = 12
let hourInterval = 3
for tickMark in stride(from: 3, through: hours, by: hourInterval) {
    print("\(tickMark), ", terminator:"")
}; print()

// 蛇と梯子ゲーム
let finalSquare = 25
var board = [Int](repeating: 0, count: finalSquare + 1)
board[03] = +08; board[06] = +11; board[09] = +09; board[10] = +02  // 梯子
board[14] = -10; board[19] = -11; board[22] = -02; board[24] = -08  // 蛇
var square = 0
var diceRoll = 0
while square < finalSquare {
    diceRoll += 1
    if diceRoll == 7 { diceRoll = 1 }
    square += diceRoll
    print("\(square) ", terminator: "")
    if square < board.count {
        square += board[square]
    }
}
print("Game over!")

// No ladder on the board takes the player straight to square 25, and so
// it isn't possible to win the game by moving up a ladder.
// ?
square=0; diceRoll=0
repeat {
    print("\(square) ", terminator: "")
    square += board[square]
    diceRoll += 1
    if diceRoll == 7 { diceRoll = 1 }
    square += diceRoll
} while square < finalSquare
print("Game over!")
```

```
// conditional statements 条件文
var temperatureInFahrenheit = 30                          // -1.1℃
if temperatureInFahrenheit <= 32 {
    print("It's very cold. Consider wearing a scarf.")  // 実行される
}
temperatureInFahrenheit = 40                              // 4.4℃
if temperatureInFahrenheit <= 32 {
    print("It's very cold. Consider wearing a scarf.")
} else {
    print("It's not that cold. Wear a t-shirt.")        // 実行される
}
temperatureInFahrenheit = 90                              // 32.2℃
if temperatureInFahrenheit <= 32 {
    print("It's very cold. Consider wearing a scarf.")
} else if temperatureInFahrenheit >= 86 {
    print("It's really warm. Don't forget to wear sunscreen.")  // 実行される
} else {
    print("It's not that cold. Wear a t-shirt.")
}
temperatureInFahrenheit = 72                              // 22.2℃
if temperatureInFahrenheit <= 32 {
    print("It's very cold. Consider wearing a scarf.")
} else if temperatureInFahrenheit >= 86 {
    print("It's really warm. Don't forget to wear sunscreen.")
}                                                         // いずれも実行されない。

// switch
// Every switch statement must be exhaustive. switch 文は必ず網羅的にする。
var someCharacter: Character = "z"
switch someCharacter {
case "a":
    print("The first letter of the alphabet")
case "z":
    print("The last letter of the alphabet")  // 実行される
default:
    print("Some other character")
}
let anotherCharacter: Character = "a"
switch anotherCharacter {
```

```
case "a", "A":
    print("The letter A")  // 実行される
default:
    print("Not the letter A")
}

// interval matching 区間マッチ
let approximateCount = 62
let countedThings = "moons orbiting Saturn"
let naturalCount: String
switch approximateCount {
case 0:
    naturalCount = "no"
case 1..<5:
    naturalCount = "a few"
case 5..<12:
    naturalCount = "several"
case 12..<100:
    naturalCount = "dozens of"  // 実行される
case 100..<1000:
    naturalCount = "hundreds of"
default:
    naturalCount = "many"
}
print("There are \(naturalCount) \(countedThings).")

let somePoint = (1, 1)
switch somePoint {
case (0, 0):
    print("\(somePoint) is at the origin")
case (_, 0):                                  // _はワイルドカード
    print("\(somePoint) is on the x-axis")
case (0, _):
    print("\(somePoint) is on the y-axis")
case (-2...2, -2...2):                        // 区間マッチと併用できる
    print("\(somePoint) is inside the box")  // 実行される
default:
    print("\(somePoint) is outside of the box")
}
```

```
// value binding 値束縛
let anotherPoint = (2, 0)
switch anotherPoint {
case (let x, 0):                                        // 第 0 要素を記号 x に束縛する
    print("on the x-axis with an x value of \(x)")  // 実行される
case (0, let y):
    print("on the y-axis with a y value of \(y)")
case let (x, y):
    print("somewhere else at (\(x), \(y))")
}

// where clause
let yetAnotherPoint = (1, -1)
switch yetAnotherPoint {
case let (x, y) where x == y:                       // where 節で制約を加える
    print("(\(x), \(y)) is on the line x == y")
case let (x, y) where x == -y:
    print("(\(x), \(y)) is on the line x == -y")  // 実行される
case let (x, y):
    print("(\(x), \(y)) is just some arbitrary point")
}

// compound cases
// 実行される本体の処理を共有するケースらは「,」で並べる。
someCharacter = "e"
switch someCharacter {
case "a", "e", "i", "o", "u":
    print("\(someCharacter) is a vowel")  // 実行される
case "b", "c", "d", "f", "g", "h", "j", "k", "l", "m",
     "n", "p", "q", "r", "s", "t", "v", "w", "x", "y", "z":
    print("\(someCharacter) is a consonant")
default:
    print("\(someCharacter) is not a vowel or a consonant")
}

// 複合ケースと値束縛を同時に使うには、すべてのケースで同じ型を束縛すればいい。
let stillAnotherPoint = (9, 0)
switch stillAnotherPoint {
case (let distance, 0), (0, let distance):
    print("On an axis, \(distance) from the origin")  // 実行される
```

```
default:
    print("Not on an axis")
}

// control transfer statements 制御移転文
// とは、continue、break、fallthrough、return、throw の 5 個である。
// continue の例
let puzzleInput = "great minds think alike"
var puzzleOutput = ""
let charactersToRemove: [Character] = ["a", "e", "i", "o", "u", " "]
for character in puzzleInput {
    if charactersToRemove.contains(character) {
        continue  // 特定の文字は無視する
    }
    puzzleOutput.append(character)
}
print(puzzleOutput)  // -> "grtmndsthnklk"

let numberSymbol: Character = "三"  // Chinese symbol for the number 3
var possibleIntegerValue: Int?      // デフォルト値 nil
switch numberSymbol {
case "1", "  ", "一", "  ":     // アラビア数字、アラビア語数字、中国語、タイ語
    possibleIntegerValue = 1
case "2", "  ", "二", "  ":
    possibleIntegerValue = 2
case "3", "  ", "三", "  ":
    possibleIntegerValue = 3  // 実行される
case "4", "  ", "四", "  ":
    possibleIntegerValue = 4
default:
    break  // 網羅的にするために default ケースを設け空にしないために break を書く。
}
if let integerValue = possibleIntegerValue {  // optional binding
    print("The integer value of \(numberSymbol) is \(integerValue).")  // 実行される
} else {
    print("An integer value could not be found for \(numberSymbol).")
}

// fallthrough
let integerToDescribe = 5
```

```
var description = "The number \(integerToDescribe) is"
switch integerToDescribe {
case 2, 3, 5, 7, 11, 13, 17, 19:
    description += " a prime number, and also"  // 実行される
    fallthrough
default:
    description += " an integer."               // 実行される
}
print(description)



square=0; diceRoll=0
// gameLoop: while square != finalSquare {
gameLoop: while true {  // ループ条件を与えなくても動作は同じ。
    diceRoll += 1
    if diceRoll == 7 { diceRoll = 1 }
    print("\(square + diceRoll) ", terminator: "")
    switch square + diceRoll {
    case finalSquare:
        break gameLoop     // switch ではなく gameloop を break する。
    case let newSquare where newSquare > finalSquare:  // where 節用の値束縛
        // continue gameLoop  // ラベルを与えなくても動作は同じ。
        continue
    default:
        square += diceRoll
        square += board[square]
    }
}
print("Game over!")

// guard 文
func greet(person: [String: String]) {
    guard let name = person["name"] else {  // 束縛は以降で有効
        return  // guard のボディは fallthrough を許されない。
    }
    print("Hello \(name)!")
    guard let location = person["location"] else {
        print("I hope the weather is nice near you.")
        return
```

```
    }
    print("I hope the weather is nice in \(location).")  // optional 束縛を利用する
}
greet(person: ["name": "John"])
greet(person: ["name": "Jane", "location": "Cupertino"])

// checking API availability
if #available(iOS 10, macOS 10.12, *) {
    print("Use iOS 10 APIs on iOS, and use macOS 10.12 APIs on macOS")
} else {
    print("Fall back to earlier iOS and macOS APIs")
}
```