Diving in with Swift (Part 1)

If you're short on resources for the Swift 5.X language and wish to have a comprehensive outline of it, visit the official documentation located at the following link:
https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html

Note that Swift is available on Linux if a Mac is unavailable to you, but you will not be able to develop Apple platform apps natively without one.[1] [2]

1. What's the purpose of having a constant versus a variable? When would you declare each, and show how to do so. (Note: When you declare a variable which you never change in your code, Xcode will issue a warning stating it's best to be a constant).

2. In other languages, we are used to certain data types being immutable or mutable. In Swift, this behavior is defined differently. As mentioned above, how do you differentiate between the two constructs? For example, suppose I want: a mutable string, and an immutable string; give two declarations (and initializations) of both.

3. Swift allows you to declare variables which are both type-inferred and explicitly typed. Although this is allowed, what is the preferred practice? Below are some explicitly-typed variables/constants. Change their declarations to match the Swift best-practice:

```
var a: Int = 3
var b: Int = 9
var c: String = "Hello from Swift!"
var d: Float = 45.0
var e: Int = Int("4")
let pi: Double = 3.141596
```

---

[1] Apple platforms include: iOS, macOS, tvOS, watchOS, and iPadOS. To use Swift on Linux, visit the official Swift website as linked to download it for your respective operating system.

[2] If your computer doesn't support Swift 5 and up, you can still use the language, but the syntax may be slightly different in solutions and potentially this practice suite. This was written using primarily Swift 5.X.

4. (True / False) Swift allows direct comparison between strings:

   EX: `"Str 1" == "Str 2"`

5. There are multiple ways of performing string concatenation. You are free to use the "+" operator. For starters declare a few strings and begin to concatenate them together or to any other string you'd like. The restrictions are that you must use all of the following: String interpolation, the addition operator, shorthand assignment (i.e. +=), and lastly existing methods as applicable to string concatenation.

6. Using ONLY string interpolation, create a string variable containing a quoted phrase, then place it within another string. Below is an example:

```
let quote = "Do or do not, there is no try"
```

expected output for example:

```
"Yoda said to the young padawan, 'Do or do not, there is no try'."
```

7. Suppose I have the following constant declaration. Can I perform the following tasks, and if not, why? How can you fix it? There are two ways, name both.

```
let myStr = String( )
myStr += "GIVE ME FORTUNE!"

let myStr2 = "I am not the "
myStr2.append(" center of attention!")
```

8. Similarly to C, you can construct strings out of an array of characters. We will explore arrays in a few sections, but take the below character array and create a string from it:

```
let atHomeEP: [Character] = ["B", "r", "e", "a", "k", "!"]
```

9. Strings have many properties such as count, isEmpty, etc. Just as well, they can be indexed like any array, as long as the subscript contains a String.Index value. You can obtain a String.Index by using the `.index()` method. For this especially, I'd recommend using playgrounds or the documentation to see all the possibilities for index. Below is a small example:

```
let str = "My str"
let myInd = str.index(str.startIndex, offsetBy: 4)
```

   Create a string by indexing two or more characters of your choice.

10. Below I have a string that I wish to augment. You'll see two versions: a before and after; your task here is to make the ***same*** string into the new one.

```
Before: "Oh no, I can do this"
After:  "Oh no! I can do this."
```

11. Same as Q.10

```
Before: "Oh no, I can do this"
After:  "Oh no, I'm sure I can do this."
```

   For this one, though, return the new string to the original by removing the additions.

12. Here's a fun trivia question. When you create substrings in Swift, how is the memory handled? Keep in mind, there are a few nuances. That is, it conservers memory cost unless you do something to it.

13. Using an obtained index, return a substring of any string using RANGE syntax. This is similar to python list slicing. For instance: `myArr[1:]`, in python would return the values from index 1 to the end. In Swift, however, this is a ***closed range***. That is [-, +] instead of [-, +).

14. As a continuation of the above question, I will give a single string. You will find the specified substrings using indexes and the range syntax.

```
let str = “I would have stayed at home!”
a. “would have”
b. “home!”
c. “I would stay home!”
d. “I have a home!”
e. “stayed”
f. “at”
g. “I would stay!”
h. “I should have a home!”
```

15. Do one final substring set using the prefix and suffix methods.

Strings have many more available features and the above only scratched the surface. For example, you can use emojis, or the hasPrefix and hasSuffix methods for fast substring matching.

16. (True / False) Swift allows you to use zero- or “null”-like values as part of booleans? For instance, in C it would be valid to say something like: `if ( myList ) { … }` to test if a value is non-zero or not null. If False, what does Swift allow?

17. Now we fall into the discussion of optionals. What is the purpose of having optionals within your code? Are they necessary to have all the time, and if not, why?

18. (True / False) You cannot have variables declared without an initial value. There is a technicality here, though - what is it? Reminder that we have the noted best-practice for swift variables and constants, but it’s not *always* applicable.

19. How can you get around the previous problems? There are two ways you can do so. One involves having the possibility of a nil value, and the other disallows nil values. This will open the ability to assign a value later in your code.

20. We have the following snippet of code. As it is, Xcode generates an error. I want the stored value to be a guaranteed numeric value. How do I fix this?

```
var num: Int? = nil
let speedOfLight = 300_000

if speedOfLight < 900_980 {
    num = 80
} else {
    num = 40
}
print(num + 300) // error happens here
```

21. When do you use optional unwrapping? And when does unwrapping cause a crash in the application? Think about what you're designating to the compiler when you perform an optional unwrap.

22. Of course, there are many ways to deal with optionals and the potential of a nil return value. While unwrapping is fine, it's not the safest way to use optionals. As it stands, there are the following methods: Optional unwrapping, Conditional guarding, Optional chaining, Optional binding, and Nil-coalescing.

    22a. When do you want to use optional "force" unwrapping?

    22b. Examine the following code. Utilizing ***conditional guarding***, translate the given code.

```
var multiplier: Int? = nil
let base = 2
...
multiplier = 50
var res = multiplier! * base
print("Multiplying \(base) by \(multiplier!) is \(res)")
```

    22c. Using ***optional binding***, translate the same code.

22d. How does optional chaining work? For instance, what happens when one of the calls in the chain is a nil result? Suppose you have the following code. Using optional chaining, use the “count” property to assign to a variable.

```
let myStr: String? = “Let’s try this out!”
let strSize = < your code here >
print(< your code here >) //print the strSize
```

22e. (True / False) Since the optional chaining has the potential of returning `nil`, we can use them within optional binding.

23. When do we have to specify that a variable is optional? For example, if I call a method such as ‘`readLine()`’, it returns a ‘`String?`’, which means that the type at that point is inferred. But If I simply declare a variable with a `nil`, I get an error. Why?

24. Examine the following code. Fix it or fill in the blanks to make it functional (remember that you must unwrap unless using optional binding):

```
let multi = 8
let block = true
var num: Int?

if block {
   num = nil
} else {
   num = 20
}
if num != nil {

   multi *= 9
   num *= multi

} else {

  print(num! * multi)
}
```

25. What is a better way to rewrite the above code using optional binding? Observing the final result, what could you surmise is the benefit of optional binding as opposed to conditional guarding?

26. When doing conditional guarding we are free to compare our optional variable to a `nil` or value which matches the unwrapped type. While the first comparison is legal, why does the second produce a warning? How do you fix it?

```
var myNum: String? = “some string”
if myNum == “Some string” { ... }

if myNum! == nil { ... }
```

27. Examine the following code snippet. Your task is to translate the code to solely use optional binding. The second requirement will be compacting the code to be a single optional binding statement. (See Q. 28 code snippets for clarification)

```
var num: Int? = 98
var s: String? = “s”

let stringCnt: Any? = s?.count //optional chaining can be nil!

if ( num != nil ) && ( stringCount != nil ) {
    print(“Num \(num!) with \(s!) having (stringCnt!) chars”)
}
```

28. Using a comma separator in optional binding is similar to using the '&&' operator within a conditional statement. Take a look below and determine which statements succeed and which don't.

```
var a: Int? = 98
var b: Double? = nil
var c: String? = "I am C"

if var a = a, let b = b {...}
if var a = a, let c = c {...}
if let c = c, let b = b {...}
if let c = c, var a = a {...}
if var a = a, var b = b {...}
```

29. What is the problem with the following code? How do you fix it? Note the important thing is understanding the binding here (and why it happens). Reminder: Binding creates a local, temporary variable, within the scope of the conditional; therefore, after assignment, it takes hold of the value.

```
var num: Int? = 98

if let num = num, let num2 = num {...}
```

30. Here is another case of this. Why can't I do what I want to? How could I fix it?

```
var s: String? = nil
if let str = s, let s = str?.count {
  print("\(str) has \(s) characters")
}
```

31. After all these techniques with dealing with optionals, now we have another, final technique (possibly for now): Nil-coalescence. As a reminder consider the ternary operator first.

(< condition > ? < first branch > : < second branch >)

Which is equivalent to:

```
if condition {
  // first branch
} else {
  // second branch
}
```

The nil-coalescence operator takes the following form:

```
a ?? b
```

Effectively, this is what it can be translated to:

```
( a != nil ? a! : b )
```

Which is finally (in a pseudocode way):

```
if a != nil {
   a!
}  else  {
   b
}
```

31a. Observe the following code. Translate the given code into: ternary operator, optional binding and then conditional guarding.

```
var num: Int? = 98
var res: Any = num ?? “No Dice! 😕”
```

31b. Translate the following code into nil-coalescence.

```
enum Style: String {
        case Earth = “Earth Style”
        case Water = “Water Style”
        case Fire = “Fire Style”
        case Metal = “Metal Style”
        case Lightning = “Lightning Style”
}

var preferredStyle: Style
var userStyle: Style? = Style.Lightning
var defaultStyle = Style.Water

 if userStyle != nil {
    preferredStyle = userStyle!
} else {
    preferredStyle = defaultStyle
}
```

31c. As with optional binding, there is a reason to use nil-coalescence. What is it here? It would seem pretty apparent. Why in the above code, did I unwrap the userStyle when assigning to the preferredStyle?

31d. Instead of unwrapping userStyle, could I make preferredStyle an optional? If so, what would I need to use the value assigned to preferredStyle later?

31e. Rewrite the given code above to use only optionals with the exception of defaultStyle.

32. We have the following code. There are a few issues. Resolve it using unwrapping, then resolve it using optional binding. Identify why a particular problem exists - in essence note the difference between optionals and regular variables.

```
var a: Int? = 86
var b = a

var s1 = “S1”
var s2: String? = “S2”

var specialStr: String

var s3: String? = “S3”
var s4 = s3
s2 = s3
specialStr = s3
```

33. What are the rules to use optional chaining? (i.e. when can/can’t you use it).

34. What are the rules to use optional binding?

35. This was asked earlier, but it’s worth asking again. When is it best to use an optional instead of simply declaring the variable/constant with it’s type? (i.e. using the ‘?’ vs not using it) - This is one of the most important concepts in Swift.

36. This one is interesting, and is similar to how substrings work. When you have two variables sharing a reference, what happens when you change one variable’s value? That is, if you have a string and another reference to it, for instance, what happens if you decide to append something to either one? Is the reference still shared? If not, why?

37. What is the purpose of utilizing enumerations in a programming language? What is the key difference between using enums in C versus using them in Swift? As a hint consider that Swift’s enums support class features such as: computed properties, instance methods, etc.

38. Given the above, what is the single restriction that enums have? What is the general syntax of an enum? Feel free to use a grammar-like structure to describe it, or even provide an example enum if you wish. Since an example can be seen below, provide two examples: one with and one without a type declaration.

39. What is a raw value and what is a member variable? When can you use raw values within an enum? So could I always do something like: `myEnum.myValue.rawValue`?

40. Determine which of the following are member values and which are raw values.

```
enum Instruments: String {
      case String = “Piano”
      case Wind = “Flute”
       case Percussion = “Drums”
}
```

41. What kind of raw values are allowed in enums? For example, are you allowed to use collections in enumerations, or just certain data types?

42. Although somewhat different, Swift shares some similarities with the enums presented in C. Name one specific similarity and provide an example. Hint: C only supports integral (numeric) enums.

43. Enums and type inference are incredibly powerful. After declaring an enum type, and using the above enum, (true / false), the following is a valid piece of code. And if so, what is an equivalent way you could write it?

```
var inst = Instruments.Strings
inst = .Wind
```

44. (True / False) The raw values within an enum and the respective member values are two distinct data types?

45. Explain what the following piece of code is doing and what, if any, the differences are between the statements you see.

```
var someInst = Instruments.Strings
someInst = .Wind
type(of: someInst)

switch someInst {
    case .Wind where Instruments.Strings.rawValue == "Guitar" :
        print("hi")
    default:
        print("hi2")
}

if someInst == .Wind && Instruments.Strings.rawValue == "Guitar"
{
    print("yes")
}
```

46. Ignoring the 'where' clause for now, what is another way I could have written the condition after it? Note that someInst has access to the same 'rawValue' property. Assume we don't reassign someInst.

47. In some cases, we don't have access to the rawValue property. When does that happen?

48. Declare an enum called 'Ranks' which uses numbers as raw values to indicate Bronze, Silver, Gold, and Platinum in that order. You don't need to assign all of the raw values, just make sure Bronze is set to 1 since Swift will automatically assign the rest.

49. Using this enum, and the following array, you will assign a string to a specific feedback response string, if it is found. If a certain response is NOT found, you will assign nil to this string.

```
let rates = [“Great!”, “Poor”, “It was ok”, “Just good”]
```

50. Since there is no default case for the enumeration, you will also ensure that the enum variable for the ranking system is capable of handling a nil value.

51. Iterating through our ratings, we want to find any string which contains the word “amazing”. If we find any, then we can start off the rating as Platinum. If not, then we have to start our rating at nothing.

52. Likewise, we wish to iterate through our array and see if there’s anything with the following keywords: ok, good, poor. For every match, we keep a tally and that will be used as the final score. So, if it is a score of 2, then it would be equivalent to Silver, and so forth.

53. At the end, print out a statement as follows: “This restaurant received a score of (Bronze/Gold/Silver/Platinum)”. DO NOT hard code the score - there is a way to do it using only the variable.

54. Declare an enum called ‘Majors’ with the following abbreviations: CS, MUS, SOCS, PSY, ECO, BUS; each having the following raw values, respectively: “Computer Science”, “Music Studies”, “Social Sciences”, Psychology”, “Economics and Finance”, “Business Studies”. Play around with a few variable/constant declarations. Given the following function outline, print out the statements that are commented. Use both if-statements and switch-statements to try it out.

```
func majorOpinions(major: Majors) -> Void {
    // CS: Computer Science is the best!
    // SOCS: Social Sciences is hard!
    // MUS: Music Studies is cool!
    // PSY: Psychology is interesting!
    // ECO: Economics and Finance is FANCY.
    // BUS: Business Studies is popular…
    // I don’t know what major that is 😕
}
```

Just as with the ratings problem, don’t hard code the abbreviations. You can show it with the var.

55. (True / False) Enums with raw values can also include member values with associated values?

56. Using your previous answer, is there anything wrong with the following code? If yes, what is it?

```
enum Languages: String {

  case German = “DE”
  case English = “EN”
  case French = “FR”
  case Korean(String, String, String)

}
```

57. Given that enums don’t support raw values which aren’t assignable as literals, we can’t use collections or anything aside from the basic “primitives” we are used to (I use primitives loosely). That being said, what could you conclude is a use for enums with associated values? Keep in mind, that you can think of this as a mini-constructor within a class. This means that these values can be anything.

58. Remember that enums effectively serve as templates to be used as types later on. That being said, suppose you have an inventory which conforms to a specified set of rules such as name, price, item count. Create an enum which has the following members conforming to this idea: Book, Computer, Phone, Television, Headphones. Furthermore, create some variables/constants and then unpack these values to display what each item has assigned to it. Note, there are a few ways you can unpack it technically, but make sure that here it is exhaustive.

59. If you’re familiar with collections in languages like Python and Java, the ones in Swift behave similarly. We will begin with arrays! As in the best practice, suppose we wish to initialize an array of integers pre-populated; give an example.

60. Now suppose that we wish to declare a single-dimensional array without an initial value. Naturally, we could go the two routes of declaring the array with the type as an optional or not (i.e. `var myArr: [Int]` or `var myArr: [Int]?`). But, this causes some issues. Namely, the compiler knows the type the variable will *eventually* be, but what's the problem here? Provide two identical alternatives. Hint: It's similar to calling a constructor and one of the alternatives utilizes generic syntax.

61. In addition, how would I use something like the optional version of myArr? Specifically, if I'd want to append a value to this array how would I do so? Why do I have to do it this way? Furthermore, if I used said append method, would I have to store this in a variable/constant, or is nothing specifically returned from it? (Reminder that with optional chaining is useful when accessing a potentially `nil` variable/constant's properties, methods, etc).

62. Given an array as follows, what is the issue? Remember that type inference applies to homogenous values.

```
var mixyArray = [“Hello there”, 4, “Friend”, 8.19]
```

63. Following your response from the previous question, what is the rule for explicitly declaring the array types? Provide a syntactical fix for it, and describe why you can't use 'AnyObject' here.

64. Provide equivalencies for the following array declarations.

```
var a = Array<Int?>()
let b = [Int]()
let c = [[[Double]]]()
var d = Array<Array<Double>>()
let e = Array<[Int]>()
```

65. Unlike the arrays with heterogenous data, do you have to declare the array type if it contains nil values? That is, an array of optionals?

66. Declare a 2D array. The inner array values can contain ***heterogenous*** data.

67. Declare a type-inferenced 2D array. The inner array values are strictly homogenous optional data (so be sure to put in nil values). Your data may be any type you wish.

68. What would the array from Q.67 look like if you explicitly named the type?

69. Unlike with strings, you don't need to have an Index value to index arrays. They provide random access, so you're welcome to index them like normal; however, Swift is unique like Python where you can access a range using a list-splicing-like syntax. Provided the array below, and using ranged-indexing return the specified sub-values.

```
var myArr = [1, 3, 4, 5, 6, 199, 20, 12, 13, 10, 33, 5]
```

a. [1, 3, 4, 5]
b. [199, 20, 12, 13]
c. [12, 13, 10, 33, 5]
d. [1, 3, 4, 5, 6, 199, 20, 12, 13, 10, 33, 5]
e. [1, 3, 4, 5, 6, 199, 20, 12, 13]
f. [4, 5, 6, 199, 20, 12, 13, 10, 33, 5]

70. (True / False) Array slicing using the range (i.e. the '…' operator) will return an inclusive range. For instance, myArr[0…1] would be [0, 1] instead of [0, 1) like in Python with myArr[0:1].

71. (True / False) You can return the reverse of the array using slicing like in Python.

72. What is the benefit of using generics? Specifically, what is the benefit as it applies to arrays? Reminder that it's very important if using closures or instances where you are iterating through it.

73. As with arrays in python, you can insert/append values in many ways. Take the following array and show all the ways you can perform list concatenation (that includes a variation of the insert method).

```
var arr1 = [1, "Hello".hashValue, 3, 12, 22]
```

74. (True / False) Initializing an ‘Any’ array prevents using arrays as they normally would be.

75. Give the declaration that prevents the previous issue.

76. (True / False) Optional chaining is not necessary on the last call in the chain.

77. We will be making a mini regular FIFO queue. For starters, declare an array of integers. Create 10 random integers between 0 and 8 and put them in the array. Go through this array and *remove* the first element until the array is empty. Use the following for-in loop template to get started. (The ‘_’ is typically used to represent an invisible and unused variable). Please print the values as you remove them.

```
for _ in 0..<10 {

    let ran = ... // your code in this block

}
```

78. How does the insert method work? For instance if I had the following array and this method call: `myArr.insert(8, at: 2)` what would be the final result?

```
var myArr = [ 0, 1, 2, 3, 4, 5]
```

79. We will again be making a mini data structure. This time, a stack! Since arrays provide the ability to “pop”, let’s do it. Reminder: Stacks are LIFO - so the last element inserted is the first to leave. Insert as you did above (randomly), but this time don’t simply extract from the front, now you pop! Keep in mind, this will return an optional. It’s up to you how to handle this. I’d suggest optional binding perhaps.

80. What’s the difference between the ‘reversed()’ and ‘reverse()’ methods?

81. Likewise, what's the difference between the 'sort()' and 'sorted()' methods?

82. In general, explain how the closure syntax works for the sorting methods. Namely, each of the ones shown below.

```
var myArr = [ 99, 87, 10, 100, 324, 1, 0 ]
myArr.sort() { $0 < $1 }
myArr.sort() { $1 < $0 }
```

83. What's the difference between the filter method and the map method, and what is the similarity? Provide some examples of both.

84. Using the forEach method, loop through an array and print out the values.

85. As discussed earlier, homogenous data is important in certain instances where you are iterating the array. This is so you can be sure that all properties being accessed are available. Using the filter method, iterate through an array of strings and only print those with the letter 'o'. Do the same with the forEach method.

86. Translate your above iterations (Q.84 & Q.85) into the following control structures: for-in loop and while loop.

87. We will see tuples shortly, but currently use them to iterate through an array of your choice by calling the "enumerated" method. Print out the following: "At index X, array has Y stored', where X and Y are the respective values.

88. What's the problem with using the for iterators? It has to do with mutation.

89. Below are some basic dictionary declarations. Provide equivalencies as you did with arrays.

```
var dic = Dictionary<String, Int>()
var dic2 = [String: Dictionary<String, String>]()
var dic3 = [Double: Array<Int>]()
var dic4 = Dictionary<String, Double>()
```

90. We could have initialized the above dictionaries with any values we wanted to have the types be inferred instead of explicit. Do that with the above declarations.

91. Suppose you had an empty dictionary declaration (You can do any you wish). What is/are the way(s) you can add key/value pairs into your new dictionary? Hint: I can think of two - one being very python-like.

92. Why does accessing a key/value pair return an optional of the value's type? How could you handle that situation? While you can force unwrap, why would it not be a good idea with accessing a value you aren't certain exists? Use your reasoning from much earlier.

93. As with adding values, there is also multiple ways to erase a key/value pair. What are they? And why does one of them return an optional? As a reminder, nils are good indicators of a non-existent value.

94. Use both the forEach method and the for-in loop to iterate over a dictionary of your choice, printing out respective key/value pairs.

95. What does the following loop do?

```
for _ in 1... {
    print("Hello")
}
```

96. Declare an empty ***set*** that's to be filled with integers.

97. Randomly fill up your ***set*** with values.

98. Use the following sets and provide code which results in the expected outputs. NOTE: These are sets, so Swift provides specific set operations. Also, the order may be different which is okay.

SET 1: `{1, 2, 9, 10, 13, 12}`
SET 2: `{0, 2, 88}`

OUTPUT 1: `{1, 2, 9, 10, 13, 12, 0, 88}`
OUTPUT 2: `{ 2 }`
OUTPUT 3: `{ 1, 9, 10, 13, 12 }`
OUTPUT 4: `{ 0, 88 }`

99. What is the purpose of the ‘as’ keyword? Is it always necessary when we have conversion convenience initializers such as `String()`, `Double()`, `Int()`, etc.

100. The ‘as’ keyword is also known as _____ (2 words). Typically used in _____ and _____ (both 2 words). Which is why sometimes this keyword requires to be designated as an optional. Take a look at !. Q.101 if you need clarification.

101. Take a look at the following code. Why do we use ‘as?’ Instead of just ‘as’. Describe what happens at the end, and provide an equivalence using if-statements.

```
let a: NSArray = [3, 21.3, “Hello there!”]
let b = a as? [Int] ?? [1, 2, 3]
```

102. Provide an example where using ‘as’ always succeeds and doesn’t require designation as an optional.

103. We’ve seen force unwrapping as they relate to all optionals. In that case, provide an example when using ‘as!’ Is perfectly okay.

104. Below I provide an example of using the ‘as’ keyword and then ‘as!’, but there is a problem. What is it, and how can I fix it? Hint: It’s the reason we use ‘?’ In optionals - think about your response in Q.101. Then fix it!

```
let a: NSDictionary = [1:”One”,2:”Two”,”Three”:3]
let b = a as! [Int:String]
```

105. It’s cool when we can replace letters with numbers. I will provide a small list of words below that you have to build an array from. Furthermore, I will also provide key/value pairs representing letters and their respective replacements. You will iterate through your array of strings and for each word, replace all occurrences of the letters with their value in the dictionary. These changes must take place **in-place.** You are welcome to do this however you think is best.

Key/Value pairs (Do these for the lowercase versions of the letters, too):

“A” : 4, “L”: 1, “E”: 3, “T”: 7, “G”: 6, “S”: 5, “B”: 8, “O”: 0, “P”:9

Strings:

“Netflix”, “Doctor Who”, “Love, Simon”, “Star Wars”, “Harry Potter”, “iPhone”, “Pentatonix”

As a bonus, declare the dictionary but load the pairs separately. That is, don’t initialize them, but give them values after declaration.

106. What are tuples? (True / False) they allow heterogenous data, but are effectively immutable. That is, you can’t append or remove values.

107. Declare a tuple of your choice. Then, access the individual elements within the tuple.

108. Since indexing tuples is slightly different from that of python, how can we make it to where accessing the elements is more verbose? Hint: This allows us to replace the use of a class or struct if we don’t need the use of properties or methods, but just need to store data.

109. (True / False) You can logically compare tuples as long as they contain the same types and are the same size.

110. Utilizing tuples and a dictionary, iterate through the dictionary where `(k, v)` yields `myDict[k]`. It's similar to when you enumerated the array much earlier. You may notice that in your solution you probably aren't using the 'v' variable. Provide a few alternatives.

111. I have a tuple below and some trailing assignments to the values within. Change it to be a single-line assignment. This is similar, if not exactly like, unpacking variables in python's tuples.

```
var t = (3, 2, “Hello world”, “one”)
var n = t.0
var n2 = t.1
var s = t.2

Python unpacking:

py_t = (3, 2, “Hello world”, “one”)
n1, n2, s1, s2 = py_t
```

112. What are some differences between the switch statement provided in swift versus that of C? Why, in turn, is it so powerful?

113. Switch statements in other languages require their blocks of code to be followed by the 'break' statement. Why is this NOT the case in Swift? As a result this behavior can still be simulated with a specific keyword.

114. (True / False) Switch statements must *always* be exhaustive (i.e. outlining every possible branch, followed by the default case). If false, what are the exceptions, and can you provide an example?

115. There exists an extra keyword we can use when using switch-statements (and others, as will be seen), which can filter additional keywords. What is it, and how can you use it?

116. Observe the following code, what is wrong with it?

```
enum T : String {
   case a = “alpha”, b = “alpha”
   case c = “charlie”
   case d = “delta”, e = “echo”
}
```

(See next page for 117)

117. Given the following function outline, complete the function. Some of it is done for you. The main idea is to NOT use if statements as I did here, but rather come up with alternatives using the construct you identified in Q. 115. Note that this switch statement as-is will not work without the modification! I provide a sample function call right below.

```
func swimmingInCode(tup: (name: String, Int), lock: Bool?) -> Void {

    var lock = lock ?? true

    let ns = [“Scott”, “Mitch”, “Matt”, “Kirstin”, “Kevin”]

    switch tup {

      case let (str, age):
        if (str == “Scott”) && (lock != true) {
            print(“Got \(str) who is \(age) years old.”)
        }
      case let (str, age):
        if (str == “Mitch”) && (lock != true) {
            print(“Got \(str) who is \(age) years old.”)
        }
      case let (str, age):
        if (str == “Matt”) && (lock != true) {
            print(“Got \(str) who is \(age) years old.”)
        }
      case let (str, age):
        if (str == “Kirstin”) && (lock != true) {
            print(“Got \(str) who is \(age) years old.”)
        }
      case let (str, age):
       if (str == “Kevin”) && (lock != true) {
            print(“Got \(str) who is \(age) years old.”)
       }
      default:
       if !ns.contains(tup.name) {
           print(“Unknown name or locked!”)
       }

    }

}

let t = (“Scott”, 31)

swimmingInCode(tup: t, lock: nil)
```

118. Oops! This is embarrassing… The above code contains a few software engineering mistakes. Do you notice a particular pattern? Of course, the purpose of the last question was to use a special filtering construct. For this question, rewrite the switch statement to make this better code. Remember the golden rule: DRY!!! (Don't repeat yourself)

119. Given this new improvement, is a switch statement really necessary here? Can you rewrite the function to be slightly simpler?

120. Below is an enum. Use a switch statement to unpack them, but use a clause to filter the case statements. The strings must contain the letter "o" or "a".

```
enum Devices {

    case iPhone(String, String, Int)
    case iPad(String, String, Int)
    case Mac(String, String, Int)

}

let phone = Devices.iPhone("11 Pro Max", "A1203", 1000)
let tablet = Devices.iPad("iPad Pro 11\"", "B123",
1022)

let computer = Devices.Mac("Mac Pro", "C891", 5000)
```

121. What's the difference between Swift's repeat-while loop, vs many other languages' do-while loop? What is the similarity if any?

122. With switch statements, we can use the ‘fallthrough’ keyword to explicitly state we want a fall through to take place. While we could do this, if multiple cases are to match a particular block of code, we can chain them. Translate the below code to take advantage of this.

```
var someNum = 90

switch someNum {

    case 0..<10:
      fallthrough
    case 10..<100 where someNum % 2 == 0:
      fallthrough
    case 200...500:
      print(“This is a nice number!”)
    default:
      print(“This is an uncool number!”)

}
```

123. Trace the following code. What is the execution going to be like?

```
var student = (name: “Ben”, gpa: 3.14)

switch student {

    case let (name, gpa) where gpa >= 3.8:
      print(“\(name) has an excellent gpa of \(gpa)”)
    case (let name, let gpa) where gpa < 3.2:
      print(“\(name) can’t get this scholarship.”)
    default:
      let (name, gpa) = student
      print(“\(name) has a decent gpa of \(age)”)

}
```

124. By now you have seen many ways to filter your code with different clause conditions. ‘Where’ is similar to the declarative syntax in SQL, which acts sort of like the logical “&&” operator as an extra stipulation to your control flow. Effectively, it in itself acts as an ‘if’ statement. Using the following code, translate it to use the ‘where’ clause instead to behave the same way.

```
let ts = [(“One”,1),(“Two”,2),(“Three”,3),(“Four”,4)]

for t in ts {
   if t.1 % 2 == 0 {
     print(“\(t.1) is worded as \(t.0)”)
   }

}
```

125. In switch statements, the where clause may seem more useful since they provide an additional condition to filter the matching case. Why do we use it in a situation like above? Can you think of conditions it is more suited for otherwise?

126. Again, there are very many different ways of filtering your collections, some situations warranting each technique in a different way - possibly. You know you can use the for-in loop to easily iterate through collections or ranges, but you can also just as easily filter the collections and iterate that. Explain how the for-case statement works. Take a look at this piece of code:

```
let a = [(“x”, “word”), (“x”, “y”), (“x3”, “x4”)]

for case (“x”, let y) {
    print(y)
}
```

For Q.127, Q.128, and Q.129, here is a sample tuple from the required array. Note that the names array can be as large or small as you want, but must at least have three.

```
(name: “Ichigo Kurosaki”, randNum: 89)
```

127. Create an array of 101 ***named*** tuples in which each tuple must have a randomly chosen name from an array of names, and a randomly generated number between 0 and 100, inclusive. Iterate through the array and filter based off of the generated number ONLY being divisible by 2 and being larger than 0.

128. Iterate through your array again, but this time filter off of the name containing a particular letter of your choice.

129. Finally iterate through your array specifying that you want to filter based off the number being 2.

130. Observe the following code. How does it work in this particular context? Will it successfully execute the block of code?

```
var myArr = [(“Hello”, 3), (“Hi”, 5), (“Grüßen”, 8)]

if case let (“Hello”, num) = myArr[0] {

    print(num)

}
```

131. Which of the following choices reflect the equivalence of the condition in this for loop? And what does this loop do? Be sure to understand why the other choices are incorrect.

```
var myArr: [Int?] = [1, nil, 2, nil, 9, 99, nil]

for case let n? in myArr where n % 2 == 0 {
    print(n)
}
```

```
A. for case .some(let n)
B. for case .none(let n)
C. for case let n in myArr
D. None of the above
```

132. Can you rewrite the above piece of code using the if-case-let construct? If yes, do so.

133. We know that we can unpack an enum with associated values utilizing a switch statement. However, this may be necessary if we only need to unpack one particular case from the enum. Do so using the if-case-let statement with the following enum. What would be the downside of using a switch statement in this case? Rewrite this to use a switch statement for unpacking.

```
enum Businesses {

 case Accounting(Int) //where int is num of employees
 case MusicSchool(Int)
 case RecordingLabel(Int)
 case StockBrokers(Int)

}

var b1 = Business.Accounting(879)
var b2 = Business.MusicSchool(189)
var b3 = Business.StockBrokers(1777)

// your code for unpacking
```

From here forward - unless stated otherwise - when you're asked to create a function, you must create three alternatives: within the parameter list, you have to declare it 1. With argument labels; 2. Without argument labels (i.e. implicit via parameter names); and 3. Omitting the labels altogether. Note that Swift allows overloading (or rather, redeclare) of a function as long as the parameter list varies from one to another, much like Java for instance. The difference is that you can redeclare the functions using the *same* parameters as long as they vary in the ways I described with the requirements. But you can overload functions as you're used to also.

134. (True / False) When declaring a function that doesn't return anything, you don't need to specify Void as the return type, but you can?

135. (True / False) Swift allows nested functions? That is, you can declare a function within another.

136. How do Swift's variadic parameters work? Specifically, why are they so easy to use within a function?

137. Why aren't you able to mutate (change) the values of the parameters passed into a function? There are two ways to get around this. While we will see another way later, describe how you might do this with a local variable. Additionally, if you modify this local variable, does it modify the original? Why or why not.

138. Argument labels are incredibly useful, but aren't always necessary. Suppose you have the following function declaration, is the following function call correct?

```
func notTooTricky(a: Int, b: String) { ... }

notTooTricky(a: 89, b: “Testing”)
```

139. Using your answer from above, why or why not do you think the function call is correct? Furthermore, why do you think, if applicable, the labels within the function call are required in this instance?

140. Below is a function template. Complete the function to operate correctly. Whereas before there was an absence of explicit argument labels, here there are; identify them and provide a function call after the function.

```
func addEm(addNum a: Int, withNum b: Int) -> Int {



}

// call the function here
```

141. Observing your answer above, why do you think argument labels may be useful? Fun fact: originally they stem from Objective-C.

142. To demonstrate the flexibility of functions in Swift, we can also have a C-like function syntax. For instance, not needing to specify the argument labels in any context or even choose to have some not needed. For this one you don't need to have the 3 function alternatives, just create a function where each of the following function calls are legal. To be clear, the sentence-like parameters are the ***argument labels*** and are not the parameters of the function.

```
//assume that a and b are two constants

add(a, b)
add(firstVal: a, secondVal: b)
add(a, withNum: b)
add(a: a, b: b)
```

143. (True / False) You can return multiple values from a function? What are the ways you can do so, if you can?

144. What is the purpose of the guard statement? How does it work? And what are the requirements to use it?

Rewrite the following conditionals to utilize the guard statement. You may assume that the functions they would be in don't return anything.

145.

```
var a = 98
var b = 98

if (a >= 100) || (a == b) {
   print("What a success!")
} else {
   print("My condition failed 😭")
}
```

146.

```
var str: String? = “Hello guards!”

if (str != nil) && (str!.contains(“Hello”)) {
    print(“This is the string you’re looking for!”)
} else {
    print(“OH NO! STR IS NIL!”)
}
```

147.

```
var str: String? = “Hello guards!”

if let _ = str , let _ = str?.contains(“Hello”) {
    print(“This is the string you’re looking for!”)
} else {
    print(“OH NO! STR IS NIL!”)
}
```

148. (Bonus: True / False) The below code is equivalent to Q.147

```
var str: String? = “Hello guards!”

if let _ = str?.contains(“Hello”) {
    print(“This is the string you’re looking for!”)
} else {
    print(“OH NO! STR IS NIL!”)
}
```

149. Create a function which takes three optional integer parameters. The third must be a variadic parameter. Reminder that you have to create three variations of this function as required above. Additionally, protect the function from using the first two nil parameters using both a guard statement and conditional-style optional-binding. The function must output the following for each of the variadic values:

```
"x + y * z = result"
```

Make sure you use case filtering for the variadic parameters in both styles.

150. Create a function for calculating fibonacci numbers. It will take in one integer value and the limit will be 35 without DP (to avoid a ridiculous number of recursive calls). Feel free to use guards or simply if-statements for your base cases; however, protect the function from a number > 35 using both guard and if-statements.

$$F_0 = 0$$
$$F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2}$$

151. Create a function for calculating fibonacci numbers. This time, you will use bottom-up DP (start from bottom using tabulation).

152. Create a function for calculating fibonacci numbers using top-down/ memoization DP.

153. Given the following function declaration, which of the function calls afterwards are valid?

```
func callDef(_ a: Int?, withDefault b: String = “def”)
{

    guard let a = a else {
      print(“first val is nil, with b valuing: \(b)”)
      return
    }

    print(“First val is \(a) with b valuing \(b)”)

}


callDef(nil)
callDef(nil, withDefault: “30”)
callDef(3)
callDef(90, withDefault: “40”)
```

154. Create a function which takes in an optional array and a number to look for. First, make sure to protect the function from a nil value. Then, create a nested function to check if said array is sorted increasingly. If not, sort it in-place. Then perform a binary search on this array using an iterative approach. The function will return nil if the value is not, otherwise it will return the index at which it is found.

155. Do the same as above except find a way to perform the binary search recursively. I’d suggest using nested functioning if possible.

156. Create a function which takes in a dictionary and a id to look for. The keys of the dictionary will be a string: abc123, and the values will be a tuple: (name, gpa, major). The function will act as a makeshift database. So, you will look to see if the id given exists. If not, the function will print out “Student X is not enrolled”. Then, the list of enrolled students will be printed and false will be returned. Otherwise, you will print out the data of the student and return true. As the dictionary may not exists (no one is enrolled) ensure to protect the function. You will not be required to return nil, just false as a fail-safe default.

157. This function will also take in the same type of dictionary as above. Additionally, two parameters indicating desired major to search and gpa to search. You must filter, in both cases, both parameters. Also, you have to handle what happens if the parameters are not provided (or nil), or if the major/gpa is not found. Same goes for an empty dictionary. In the end the function will return either: nil, or an array of values representing the found majors and gpa’s.

158. Create a function which takes in an array, an index for that array, and a value to place into the array. Index into this array, and replace the value with the one given (how you do this is your choice), but you must do so in-place. Additionally, if the index exceeds the limits, either negatively or positively of the array, wrap it around so that it is gracefully accessed.

159. Create a function that takes in a randomly generated array (yes you have to do that), and a “step factor” with a default value of 1. Simply do a linear iteration of the array, but every time you hit an even number, increase the step factor by one. This step factor will be used as the update statement within your iteration. Note that you can’t use inout as a requirement for this function.

160. CHALLENGE: Swift has a few ways of handling file IO. For this particular question there will be a rather easy way of doing it. For starters, create a new plaintext document and place it in your resources folder located in a playgrounds project. Simply parse it out by reading it and spitting out an array of the delimited strings via a newline.