**UNIT 2 READING GUIDE**

**Chapter 5: Control Flow**

* What is the difference between if, else, else if, ifelse, case\_when, and switch?  When might you use each?
  + If:
    - If condition is TRUE, true\_action is evaluated; if condition is FALSE, the optional false\_action is evaluated.
  + Else:
    - When you use the single argument form without an else statement, if invisibly (Section [6.7.2](https://adv-r.hadley.nz/functions.html#invisible)) returns NULL if the condition is FALSE.
  + Ifelse:
    - Given that if only works with a single TRUE or FALSE, you might wonder what to do if you have a vector of logical values. Handling vectors of values is the job of [ifelse()](https://rdrr.io/r/base/ifelse.html): a vectorised function with test, yes, and no vectors (that will be recycled to the same length)
    - Only when the yes and no vectors are the same type as it is otherwise hard to predict the output type.
  + case\_when
    - Another vectorised equivalent is the more general [dplyr::case\_when()](https://dplyr.tidyverse.org/reference/case_when.html). It uses a special syntax to allow any number of condition-vector pairs:
  + Switch
    - Closely related to if is the [switch()](https://rdrr.io/r/base/switch.html)-statement. It’s a compact, special purpose equivalent
    - The last component of a [switch()](https://rdrr.io/r/base/switch.html) should always throw an error, otherwise unmatched inputs will invisibly return NULL
* What happens if you give an if statement a vector of TRUE and FALSE?
  + The if statement will return an error because it can only accept a single TRUE or FALSE
* What is a situation where a while loop might be more useful than a for loop?
  + for loops are useful if you know in advance the set of values that you want to iterate over
  + So while loops are more useful when you don’t the entire set of values you want to iterate.
* What is the difference between next, break, and print inside a loop?
  + next exits the current iteration.
  + break exits the entire for loop.
  + print prints for each iteration inside a loop.
* What are good alternatives to using a for loop?  When might these not be sufficient or desirable?
  + A while loop or a repeat. These alternatives are much more flexible than a for loop, but they will not be sufficient when we already know what to iterate and in general should have the least flexible solution.

**Chapter 6: Functions**

* What are the formals, body, and enviroment of a function?
  + [formals()](https://rdrr.io/r/base/formals.html), the list of arguments that control how you call the function
  + [body()](https://rdrr.io/r/base/body.html), the code inside the function
  + [environment()](https://rdrr.io/r/base/environment.html), the data structure that determines how the function finds the values associated with the names.
* What is a primitive function?
  + call C code directly
  + Primitive functions are only found in the base package. While they have certain performance advantages, this benefit comes at a price: they are harder to write
* What is lexical scoping?  What does this mean about using the same names for objects inside and outside of functions?
  + Lexical scoping is the act of binding a name to a value. Here we’ll discuss **scoping**, the act of finding the value associated with a name.
  + R uses **lexical scoping**37: it looks up the values of names based on how a function is defined, not how it is called. “Lexical” here is not the English adjective that means relating to words or a vocabulary. It’s a technical CS term that tells us that the scoping rules use a parse-time, rather than a run-time structure.
  + R relies on lexical scoping to find *everything*, from the obvious, like [mean()](https://rdrr.io/r/base/mean.html), to the less obvious, like [+](https://rdrr.io/r/base/Arithmetic.html) or even {. This gives R’s scoping rules a rather beautiful simplicity.
* R’s lexical scoping follows four primary rules:
  + Name masking
  + Functions versus variables
  + A fresh start
  + Dynamic lookup
* What do we mean by R having "first-class functions"?  Why is this useful?
  + It’s very important to understand that **R functions are objects** in their own right, a language property often called “first-class functions”. Unlike in many other languages, there is no special syntax for defining and naming a function: you simply create a function object (with function) and bind it to a name with [<-](https://rdrr.io/r/base/assignOps.html):
  + You have free range of what to name functions.
* If you pass in an R code expression as an argument to a function, when does the code get evaluated?
  + In R, function arguments are **lazily evaluated**: they’re only evaluated if accessed.
* How is & different from &&?
  + When evaluating empty inputs such as empty vectors, NULL, and NA, one will make a logical error while the other will evaluate as NA or FALSE.
  + The longer form is appropriate for programming control-flow and typically preferred in if clauses.
* What is "passing the dots" and why might it be useful?
  + Functions can have a special argument ... (pronounced dot-dot-dot). With it, a function can take any number of additional arguments. In other programming languages, this type of argument is often called *varargs* (short for variable arguments), and a function that uses it is said to be variadic.
* What are the four types of functions, and how are they different?

Function calls come in four varieties:

* + **prefix**: the function name comes before its arguments, like foofy(a, b, c). These constitute of the majority of function calls in R.
  + **infix**: the function name comes in between its arguments, like x + y. Infix forms are used for many mathematical operators, and for user-defined functions that begin and end with %.
  + **replacement**: functions that replace values by assignment, like names(df) <- c("a", "b", "c"). They actually look like prefix functions.
  + **special**: functions like [[[](https://rdrr.io/r/base/Extract.html), if, and for. While they don’t have a consistent structure, they play important roles in R’s syntax.

**Chapter 7: Environments**

* An environment is essentially a list of objects.  What is special about an environment that makes it more than just a list?
  + Generally, an environment is similar to a named list, with four important exceptions:
    - Every name must be unique.
    - The names in an environment are not ordered.
    - An environment has a parent.
    - Environments are not copied when modified.
* The job of an environment is to associate, or **bind**, a set of names to a set of values.
* What is *dynamic scoping*?  What consequences does this have for creating functions?
  + Looking up variables in the calling stack rather than in the enclosing environment is called **dynamic scoping**. Few languages implement dynamic scoping (Emacs Lisp is a [notable exception](http://www.gnu.org/software/emacs/emacs-paper.html#SEC15).) This is because dynamic scoping makes it much harder to reason about how a function operates: not only do you need to know how it was defined, you also need to know the context in which it was called. Dynamic scoping is primarily useful for developing functions that aid interactive data analysis,
* What is the difference between a *package environment* and a *namespace environment?*Why is it important that these are different?
  + Each package attached by [library()](https://rdrr.io/r/base/library.html) or [require()](https://rdrr.io/r/base/library.html) becomes one of the parents of the global environment. The immediate parent of the global environment is the last package you attached43, the parent of that package is the second to last package you attached, …
  + The goal of **namespaces** is to make sure that this does not happen, and that every package works the same way regardless of what packages are attached by the user.
  + Every function in a package is associated with a pair of environments: the package environment, which you learned about earlier, and the **namespace** environment.
    - The package environment is the external interface to the package. It’s how you, the R user, find a function in an attached package or with [::](https://rdrr.io/r/base/ns-dblcolon.html). Its parent is determined by search path, i.e. the order in which packages have been attached.
    - The namespace environment is the internal interface to the package. The package environment controls how we find the function; the namespace controls how the function finds its variables.
* What is the difference between a *function environment*and an *execution environment?*Why is it important that these are different?
  + Each time a function is called, a new environment is created to host execution. This is called the execution environment, and its parent is the function environment.
  + A function binds the current environment when it is created. This is called the **function environment**, and is used for lexical scoping. Across computer languages, functions that capture (or enclose) their environments are called **closures**, which is why this term is often used interchangeably with *function* in R’s documentation.

*Note: I have encountered very few R programming scenarios where it is important to understand some of the deeper structures of environments, besides what I listed above.  Feel free to dig in if you are interested, but you can skip a lot of this chapter if you want.*

**Chapter 8: Conditions**

* Give an example of when you might use an error vs a warning vs a message when designing a function.
* There are three conditions that you can signal in code: errors, warnings, and messages.
  + Errors are the most severe; they indicate that there is no way for a function to continue and execution must stop.
    - The best error messages tell you what is wrong and point you in the right direction to fix the problem. Writing good error messages is hard because errors usually occur when the user has a flawed mental model of the function. As a developer, it’s hard to imagine how the user might be thinking incorrectly about your function, and thus it’s hard to write a message that will steer the user in the correct direction. That said, the tidyverse style guide discusses a few general principles that we have found useful: <http://style.tidyverse.org/error-messages.html>.
  + Warnings fall somewhat in between errors and message, and typically indicate that something has gone wrong but the function has been able to at least partially recover.
    - Warnings occupy a somewhat challenging place between messages (“you should know about this”) and errors (“you must fix this!”), and it’s hard to give precise advice on when to use them. Generally, be restrained, as warnings are easy to miss if there’s a lot of other output, and you don’t want your function to recover too easily from clearly invalid input. In my opinion, base R tends to overuse warnings, and many warnings in base R would be better off as errors. For example, I think these warnings would be more helpful as errors:
  + Messages are the mildest; they are way of informing users that some action has been performed on their behalf.
  + Good places to use a message are:
    - When a default argument requires some non-trivial amount of computation and you want to tell the user what value was used. For example, ggplot2 reports the number of bins used if you don’t supply a binwidth.
    - In functions that are called primarily for their side-effects which would otherwise be silent. For example, when writing files to disk, calling a web API, or writing to a database, it’s useful to provide regular status messages telling the user what’s happening.
    - When you’re about to start a long running process with no intermediate output. A progress bar (e.g. with [progress](https://github.com/r-lib/progress)) is better, but a message is a good place to start.
    - When writing a package, you sometimes want to display a message when your package is loaded (i.e. in [.onAttach()](https://rdrr.io/r/base/ns-hooks.html)); here you must use [packageStartupMessage()](https://rdrr.io/r/base/message.html).
* Why might you want to use try() inside your function?  When might you prefer to use tryCatch() or withCallingHandlers() instead?
  + [try()](https://rdrr.io/r/base/try.html) allows execution to continue even after an error has occurred. Normally if you run a function that throws an error, it terminates immediately and doesn’t return a value:
  + It is possible, but not recommended, to save the result of [try()](https://rdrr.io/r/base/try.html) and perform different actions based on whether or not the code succeeded or failed49. Instead, it is better to use [tryCatch()](https://rdrr.io/r/base/conditions.html) or a higher-level helper; you’ll learn about those shortly.
  + Every condition has default behaviour: errors stop execution and return to the top level, warnings are captured and displayed in aggregate, and messages are immediately displayed. Condition **handlers** allow us to temporarily override or supplement the default behaviour.
  + [tryCatch()](https://rdrr.io/r/base/conditions.html) defines **exiting** handlers; after the condition is handled, control returns to the context where [tryCatch()](https://rdrr.io/r/base/conditions.html) was called. This makes [tryCatch()](https://rdrr.io/r/base/conditions.html) most suitable for working with errors and interrupts, as these have to exit anyway.
  + [withCallingHandlers()](https://rdrr.io/r/base/conditions.html) defines **calling** handlers; after the condition is captured control returns to the context where the condition was signalled. This makes it most suitable for working with non-error conditions.
* The handlers set up by [tryCatch()](https://rdrr.io/r/base/conditions.html) are called exiting handlers, because they cause code to exit once the condition has been caught. By contrast, [withCallingHandlers()](https://rdrr.io/r/base/conditions.html) sets up **calling** handlers: code execution continues normally once the handler returns. This tends to make [withCallingHandlers()](https://rdrr.io/r/base/conditions.html) a more natural pairing with the non-error conditions. Exiting and calling handlers use “handler” in slighty different senses:
  + An exiting handler handles a signal like you handle a problem; it makes the problem go away.
  + A calling handler handles a signal like you handle a car; the car still exists.

*If your project will involve writing functions for a package, you may want to read this chapter more deeply, especially the abort() function.  However, you can safely skip most of the deeper detail here for now.*