ST2195 Programming for Data Science
Block 04 Programming Concepts

VLE Resources

There are extra resources for this Block on the VLE – including quizzes and videos please make sure you review them as part of your learning. You can find them here: https://emfss.elearning.london.ac.uk/course/view.php?id=382#section-4

Computer Programming

Computer programming is the process of writing instructions that a computer can follow in order to perform a particular task. Programming is very much like writing down a cooking recipe: the aim is to provide precise instructions, which, if followed to the letter by someone else, the prepared meal will have the same taste and look as what you had in mind. So, as with cooking recipes, the first and most critical stage of the programming process is to be absolutely clear on what is the goal of the program or what is the problem that the program solves. In other words, to define what the program does. For example, a spreadsheet application should enable the organization and analysis of rectangular data, a word processor should make it easy to edit and format text, optical character recognition (OCR) software should be able to identify the characters in an image and extract them in a text file, and so on.

The second stage is to plan the steps that the program should take to fulfil its purpose. Typically, the set of steps takes some *inputs* (analogous to ingredients and cooking equipment) and produces predetermined *outputs* (analogous to the prepared meal). For example, the input of an OCR program is the image of a document and the output is the recognized text, possible in a text file. This sequence of steps is called a programming *algorithm*. Note here that the may be more than one algorithms that produce the same output starting from the same input, each having its own advantages and disadvantages. Typically, the aim is to produce algorithms that are fast and reliable, and require predictable time and resources to run.

Of course, the more we want the computer to do, the more instructions we need to provide. Let us come back to the recipe analogy. It is not so difficult to write a recipe for making pasta. Just boil water, throw in the pasta for 10 minutes, add salt, drain, and serve. But how about cooking curry or making a birthday cake? It definitely requires more steps and ingredients.

Avoid Spaghetti Programming

It is very important to spend some time to carefully plan your program, otherwise it is likely to create something that is difficult to understand, maintain or extend. *Spaghetti program* is a phrase commonly used to describe poorly planned or structured programs. Below is an example of a spaghetti program written in the *BASIC* language (one of the early programming languages) that first prints "This line prints first", then "This line prints second" and then ends.

```
10 GOTO 50
20 PRINT "This line prints second"
30 END
40 GOTO 20
50 PRINT "This line prints first"
60 GOTO 40
```

Clearly, this program is more convoluted than it should be. Line 10 (the first line) sends the computer to line 50 (the fifth line). Then line 50 gets the computer to print `This line prints first`. Afterwards the computer follows the command in line 60 that sends it to line 40, which in turn sends it to line 20. There it tells the computer to print `This line prints second`. After that, the computer automatically follows the command on the next line, which is line 30 and tells it to end. Despite that the program consists of six lines, the `GOTO` command gets the computer to move between lines, making it very difficult for a human to understand the flow and the purpose of this program. In cases of several hundreds lines of code this kind of coding will create programs that are almost impossible to understand or maintain, and should be avoided.

Structured Programming

It is essential to impose some structure to keep programs organized. Programs are usually divided into three distinct parts:

- Sequences
- Branches or conditional statements
- Iterations or loops

Branches and loops are examples of *control flow structures*.

Dividing a program into sequences, branches, and loops is very helpful in terms of isolating and organising groups of related commands into discrete chunks of code that can be modified without affecting the rest of the program.

Sequences

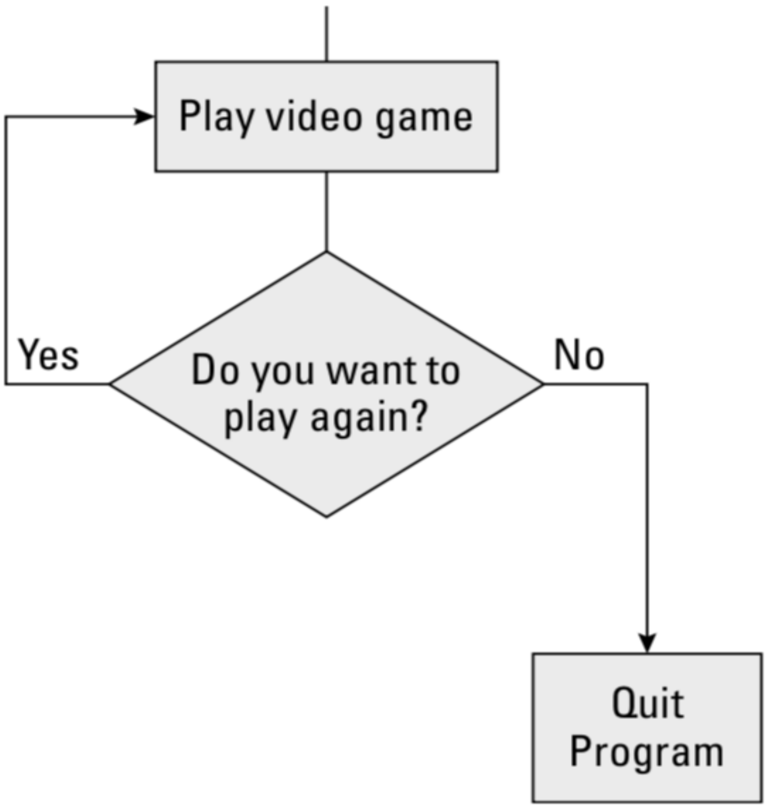
A *sequence* is a set of steps that the computer should follow one after another. An example is the following running calorie calculator that takes as inputs the user's age, weight, average heart rate and time the user ran and outputs the calories burned.

1. Calculate `[(Age x 0.2017) + (Weight x 0.09036) + (Heart Rate x 0.6309) -- 55.0969] x Time / 4.184`
2. Store the result from the calculation in 1 in `Calories Burned`
3. Print the value of `Calories Burned`

Branches

Branches consist of two or more options each of which leads to potentially different outputs. The program determines which group of commands should be followed according to whether a particular condition is satisfied or not.

For example, at the end a video game asks you, “Do you want to play again (Yes or No)?” If you choose “Yes”, the program lets you play the video game again. If you choose “No”, the program stops running, as shown in the figure below.



Branch flowchart; image taken from Wang (2008)

Iterations

Sometimes you may want the computer to run the same commands over and over again. For example, a program might ask the user for a password. If the user types an invalid password, the program displays an error message and asks the user to type the password again. If you wanted your program to ask the user for a password three times, you could write the same group of commands to ask for the password three times, but that would be wasteful. Not only would this force you to type the same commands multiple times, but if you wanted to modify these commands, you’d have to modify them in three different locations as well. Loops are basically a shortcut to writing one or more commands multiple times.

An *iteration*, or loop, consists of two parts:

- The group of commands that the loop repeats
- A command that defines how many times the loop should run

Top Down Programming

For small programs, organizing a program into sequences, branches, and loops works well. But the larger your program gets, the harder it can be to view and understand the whole thing. So a second feature of structured programming involves breaking a large program into smaller parts where each part performs one specific task. This is also known as top-down programming; i.e. designing your program by identifying the main (top) task that you want your program to solve.

For example, if you wanted to write a program that predicts the next winning lottery numbers, that is a top task of your program. Then, smaller tasks may involve

- Identifying the lottery numbers that tend to appear often.
- Picking the six numbers that have appeared most often and displaying those as the potential future winning numbers.

The idea is that writing a large program may be tough, but writing a small program is easier. So if you keep dividing the tasks of your program into smaller and smaller parts, eventually you can write a small, simple program that can solve that task. Then you can put these small programs together like building blocks, and you'll have a well-organized big program. Also, if you need to modify part of the large program, you just need to find the small program that needs changing, modify it, and plug it back into the larger program. Ideally, each small program should be small enough to fit on a single sheet of paper. This makes each small program easy to read, understand, and modify. When you divide a large program into smaller programs, each small program is a *subprogram*.

Documenting Programs and Comments

Despite the fact that most programming languages often attempt to use self-explanatory commands such as `print`,`read.csv` etc, it is often very difficult to understand what different parts of code are doing. For this reason, programmers usually add explanations directly into the code by using *comments*. A comment is nothing more than text embedded in the source code. To keep the compiler from thinking a comment is an actual command, special syntax is used to signify a comment. In R and Python all text that follows the character `#` in a line contains comments. Comments are typically short explanations that describe what the code does. Can you understand what the following sequence of R commands does and why?

```
A <- 2

B <- 3

C <- sqrt(A * A + B * B);
```

`A` and `B` are multiplied by themselves, the results are added together, and then the square root of the sum is stored in C. But this does not tell us why this code is doing this. Below we see the same set of commands but with some comments added to the code:

```
A <- 2

B <- 3

# Calculates the hypotenuse of a triangle (C) ' using the Pythagoras theorem: C2 <-
A2 + B2

C <- sqrt(A * A + B * B)
```

Even if one does not know (or care) about Pythagoras' theorem, the comments are helpful in understanding what the purpose of this line is. Another example with more comments and information is given below

```
# This formula uses the epidemic model to calculate the spread of a flu epidemic as
a function of time. P is the current population of a city, t is time measured in we
eks, c is the number of people in contact with an infected person and B is constant
value that can be determined by the initial parameters of the flu epidemic


P <- 10000

B <- 1

c <- 2000

t <- 2

F <- P / (1 + (B * exp(-c * t)))
```

Comments are therefore useful to explain both what code does and how it works. But having too many comments in a computer program is usually a sign that the code is too complicated. So, instead of writing lengthy comments, a good programmer aims to write code which is as self-explanatory as possible and uses comments only if necessary.

A program can be documented using comments that explain the purpose of one or more lines of code or of entire subprograms. Comments can also be used to provide information such as identifying the original programmer, their contact details, the license of the program, etc.

Useful Links and Resources

- Wikipedia's *Beginners' All-purpose Symbolic Instruction Code* (BASIC) language page
- Wikipedia's Spaghetti Code page

References

Wang, W. (2008). *Beginning programming all-in-one desk reference for dummies*. Wiley.

Variables, Control Flow Structures and Functions

** Note: The code chunks below should be run in the following order **

Variables

Variables provide a way to associate names with storage points in the computer that the programs can access and manipulate if needed. Variables can have different data types such as `float`, `int`, `string`, `vector`, `array`, `matrix` etc. Perhaps you are familiar with the term variable in mathematics. Programming variables are similar. A variable is a way of saving a piece of information with a specific name that we can easily reuse several times in our code. In line with the idea of variability, a programming variable allows us to easily change a value throughout our code.

Consider the following chunk of R code based on an example from Guttag (2013):

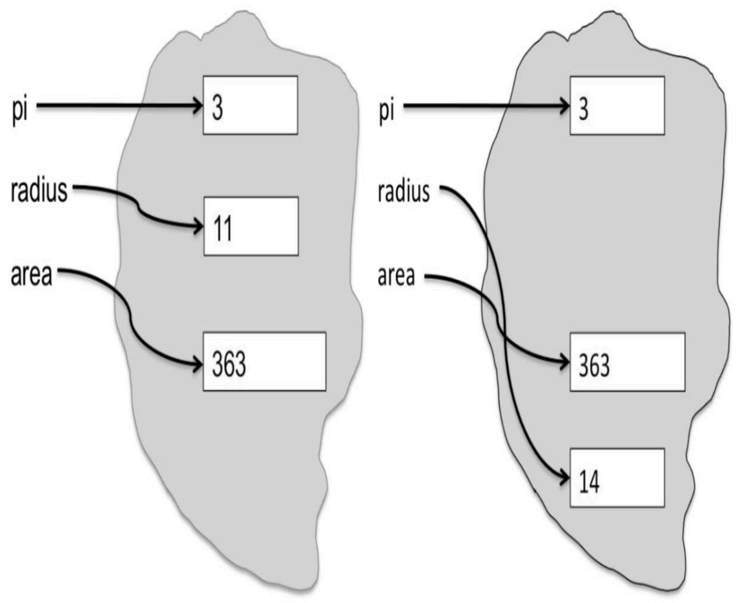
```
pi <- 3

radius <- 11

area <- pi * (radius^2)

radius <- 14
```

First, we assign the names `pi` and `radius` to different storage points that contain data of type `int` in this case. Then we assign the name `area` to a third storage point containing the value of the product between the square of `radius` and `pi`. This is represented graphically in the left panel of the figure below:



Variables assignment; image taken from Guttag (2013)

The last line of code then changes radius to another value `radius <- 14` as shown in the right panel of the figure above. Note that this assignment has no effect on the value of the variable `area` which remains `3*(11**2)`. By the way, if you believe that the actual value of `pi`, reflecting the famous number \(\pi\) is not `3`, you are right. This was done for illustration purposes only. A value of `3.14159` would offer a more accurate answer.

At this point it may be relevant to clarify the difference between variables and data. Variables reflect the assignment to storage points that contain data. So in that sense and if you are familiar with the term *random variable* in probability and statistics and the notation \(X\) vs \(x\), a programming variable refers to the mathematical function denoted by \(X\) whereas data relates the realised value of a random variable \(x\).

Note also that the names assigned are important as programs are also read by humans. It is not always easy to write programs that work correctly and programmers often devote a lot of time to understand why programs do not behave in the way they should. A good choice of variable names can provide substantial help in making computer programs easier to read. Consider the two code chunks:

```
a = 3.14159

b = 11.2

c = a * (b**2)
```

and

```
pi = 3.14159

diameter = 11.2

area = pi * (diameter**2)
```

As far as R or Python is concerned, the two code chunks do exactly the same thing. Looking at the first chunk, we see nothing wrong with the code. But a quick glance at the code in the second chunk immediately raises suspicions as to whether the program really does what the programmer wanted to; either the variable `diameter` should have been named `radius`, or `diameter` should have been divided by `2` before squaring it in the calculation of the area.

In R and Python, variable names can contain upper-case and lower-case letters and digits (but they cannot start with a digit), and the dot or underline characters. Variable names are case-sensitive.

Control Flow Structures

There are two primary tools of control flow: conditional statements (or branches) and iterations (or loops). Conditional statements allow you to run different code depending on the evaluation of a condition. Standard examples are the `if else` and `switch()` statements in R. Iterations, such as the `for` and the `while` loops, allow to repeatedly run code, potentially by changing the inputs in a particular way.

In this section we have adapted some material from chapter 5 of Wickham (2019). We strongly recommend taking a look at Chapter 5 of this resource, which is freely available online here.

Conditional Statements

It is possible to create programs that can make decisions based on the inputs you provide or the value that a variable has during run time.

With conditional statements we can write programs that determine whether a particular part of the code should be executed instead of another.

We start with the `if` statement. The basic forms of `if` statements encountered in R are

```
if (condition) true_action

if (condition) true_action else false_action
```

If the `condition` in the above code evaluates to `TRUE`, the `true_action` is performed; if the `condition` evaluates to `FALSE`, the `false_action` is performed. Of course, specifying a `false_action` is optional. If the actions are compound statements of more than one command they need to be contained within brackets `{}`

```
x <- 105

if (x > 100) {

    print("A")

    print("B")

}

[1] "A"

[1] "B"
```

We can also have nested `if` statements. For example, the following code chunk will print “A” if \(x > 90\), “B” if \(80 < x \leq 90\), and “C” otherwise (if \(x \leq 80\)).

```
x <- 80

if (x > 90) {

    print("A")

} else if (x > 80) {

    print("B")

} else
```

```
    print("C")
[1] "C"
```

Try running the above with different values for `x` to confirm that the nested if statements do what we want them to.

Note that the `condition` should be or evaluate to Boolean type, i.e. either `TRUE` or `FALSE`. Most other inputs will generate an error.

A similar conditional statement in R is the `switch()` statement. It lets us replace code like:

```
x <- "c"
if (x == "a") {
    "A"
  } else if (x == "b") {
    "B"
  } else if (x == "c") {
    "C"
  } else {
    stop("Invalid `x` value")
  }
[1] "C"
```

with the more compact

```
x <- "b"
switch(x,
  "a" = "A",
  "b" = "B",
  "c" = "C",
  stop("Invalid `x` value")
  )
[1] "B"
```

Iterations

Iterations (also known as loops) are programming structures that repeat a sequence of instructions until a specific condition is met. Programmers use iterations extensively to cycle through values, compute sums of numbers, repeat actions, and many other things. Oftentimes, a certain operation needs to be repeated many times with little variation in order to achieve a goal. Rather than copying and pasting the commands over and over again, a loop can be used instead. In addition to simplifying writing programs, loops offer another advantage. Imagine there is an error in the code you copied and pasted several times. You will then have to go and correct this error at every instance of this code. But if a loop had been used, the correction would only need to be made once!

There are three main types of iterations: the `for` loop, the `while` loop and the `repeat` loop. The `for` loops are used to iterate over items in a vector. They have the following basic form:

```
for (item in vector) perform_action
```

For each `item` in `vector`, `perform_action` is called once; then the loop moves to the next `item` of in the `vector`, and so on. For example, the following code prints all the numbers between `1` and `10`:

```
for (i in 1:10) {
  print(i)
}
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

To illustrate the points made earlier consider the naive alternative code for printing all the numbers between `1` and `10` as above.

```
print(1);print(2);print(3);print(4);print(5);print(6);print(7);print(8);print(9);pr
int(10)
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

Now imagine what would happen if we wanted to print all numbers up to `10000`; or if we accidentally copied and pasted `prin` instead of `print`. A handy rule of thumb in programming is to try to avoid copying and pasting more than twice.

It is possible to also terminate a `for` loop early. There are two ways to do it:

- `next` stops the current iteration and moves to the next `item` in `vector`.
- `break` stops the loop.

```
for (i in 1:10) {
  if (i < 3)
    next
  print(i)
  if (i >= 5)
    break
}
[1] 3
[1] 4
[1] 5
```

It is a good idea to pre-define the objects that will hold the outputs of the loop, otherwise the loop can be very slow. In the code below we begin by defining the empty vector `k` as the vector that will hold the results from each iteration of the `for` loop. Note that it has dimension `3` which the same as that of the vector we iterate over.

```
k <- numeric(3)
for (i in 1:3) {
  k[i] = i^4
}
print(k)
[1]  1 16 81
```

The `for` loops are useful if you know in advance the set of values that you want to iterate over. If you don't know, there are two related tools with more flexible specifications:

- `while (condition) action`: Performs `action` while `condition` is or evaluates to `TRUE`.
- `repeat (action)`: Repeats `action` forever (i.e. until it encounters `break`).

We can rewrite any `for` loop and use `while` instead. We can also rewrite any `while` loop and use `repeat` instead. But the converses are not true. More specifically, `while` is more flexible than `for`, and `repeat` is more flexible than `while`. It is good practice, however, to use the simplest solution to a problem, so it is usually recommended to use `for` if it suits the purpose of the program and only consider alternatives if they are really needed.

Functions

Functions are essential in splitting the program into smaller parts and allow to automate common tasks in a more powerful and general way than copying and pasting; recall the rule of avoiding copy and paste more than twice. Writing a function has three big advantages over using copy and paste:

- You can give a function a name that makes your code easier to understand.
- As requirements change, you only need to update code in one place, instead of many.
- You eliminate the chance of making accidental mistakes when you copy and paste (e.g. updating a variable name in one place, but not in another).

Writing a Function

A function can be defined with the following code:

```
func_name <- function(arguments) {
    body
}
```

The reserved word `function` is used to declare a function in R. The statements within the curly brackets form the `body` of the function. As with loops, these brackets are optional if the body contains only a single statement. Also, a function has inputs that are termed as `arguments`. Finally, the function above is given the name `func_name`.

As an example let us create a function a function called `raise()`. It takes two arguments, computes the first argument raised to the power specified in the second argument, and prints the result accompanied with a few words:

```
raise <- function(x, y) {
    # prints x to the power of y
    result <- x^y
    print(paste(x,"to the power of", y, "is", result))
}
```

The above uses the base R function `paste()` to concatenate string variables. It is good practice to check the function after writing it by calling it a few times using different arguments, and testing that the output is as expected.

```
raise(8, 2)
[1] "8 to the power of 2 is 64"
8^2
[1] 64
raise(2, 8)
[1] "2 to the power of 8 is 256"
2^8
[1] 256
```

Default Values for Arguments

It is also possible to assign default values to the arguments in a R function. For example, in the previous function we can set the default value for `y` to be `2`.

```
raise <- function(x, y = 2) {
    # prints x to the power y
    result <- x^y
    print(paste(x,"to the power", y, "is", result))
}
```

```
raise(3)
[1] "3 to the power 2 is 9"
raise(3, 1)
[1] "3 to the power 1 is 3"
```

Note that the default value of `y` is used only when no information for `y` is provided.

Composing and Nesting Functions

Now let us look at another function that converts from centimetres to inches, `cm_to_inches`:

```
cm_to_inches <- function(cm) {
  inches <- cm/2.54
  return(inches)
}
# check that the function gives 1 inch for 2.54 centimetres
cm_to_inches(2.54)
[1] 1
```

Note the in R, it is not necessary to include the return statement. R automatically returns whichever variable is on the last line of the body of the function.

In a similar manner to getting inches from centimetres, we can transform inches into yards:

```
inches_to_yards <- function(inches) {
  yards <- inches/36
  yards
}
# check that the function gives 1 yard for 36 centimetres
inches_to_yards(36)
[1] 1
```

What about converting centimetres to yards? In addition to the previous approach, we can also do it by *composing* the two functions we have already created:

```
cm_to_yards <- function(cm) {
  inches <- cm_to_inches(cm)
  yards <- inches_to_yards(inches)
  yards
}
# check that 91.44 centimetres correspond to 1 yard
cm_to_yards(91.44)
[1] 1
```

The above can also be achieved with a single line of code by *nesting* functions:

```
inches_to_yards(cm_to_inches(91.44))

[1] 1
```

This is a demonstration of how larger programs are built: basic operations are defined and then combined.

Naming Functions

As with variables it is important to remember that functions are not only read by computers but also by humans; hence their name or the comments they contain are important. Below are some good practice recommendations:

- Aim for short function names as long as they are informative of what the function does. If this is hard being informative is more important than being short, as IDEs such as RStudio typically offer autocomplete, thus making it easy to type long names.
- A frequently used convention is to use verbs for function names and nouns for their arguments although there are exceptions, e.g. `mean()` is more convenient than `compute_mean()`.

Useful Links and Resources

- Chapter 19 from "R for data dcience"
- Chapter 21 from "R dor data dcience"
- Chapter 5 from "Advanced R"
- Chapter 6 from "Advanced R"

References

Guttag, J. V. (2013). *Introduction to computation and programming using python*. MIT Press.

Wickham, H. (2019). *Advanced R*. Taylor & Francis Inc. CRC Press Inc.

Wickham, H., & Grolemund, G. (2017). *R for data science: Import, tidy, transform, visualize, and model data* (1st ed.). O'Reilly Media.

Exceptions, Error Handling and Debugging in R

Outline

This page focuses on what happens when there are errors in our code. It has adapted material from chapters 8 and 22 of Wickham (2019). The aim is to present actions for addressing the errors and the tools that are available to help. We will cover ways to fix unanticipated problems (*debugging*), show how functions can be designed to communicate problems and what actions can be taken based on those communications (*condition handling*). Finally, we will go over strategies to avoid common problems before they occur (*defensive programming*). We will illustrate concepts in R; the material of next week will focus on how to perform these actions in Python.

Debugging is the task of fixing unexpected problems in your code. Although there are some guidelines on performing it, it is usually considered an art. Of course, not all errors are unexpected. Some common problems, for example, include a non-existent file or the wrong type of input. Using *conditions*, such as *errors*, *warnings*, and *messages*, when writing functions offers a great help:

- Errors are raised by `stop()` and force all execution in a function to terminate. They are used when there is no way for a function to continue.
- Warnings are generated by `warning()` and are used to display potential problems, e.g. `log(-1)`.
- Messages are generated by `message()` and are used to give informative output in a way that can easily be suppressed by the user (see `?suppressMessages`).

Condition handling tools, like `withCallingHandlers()`, `tryCatch()`, and `try()` allow users to take specific actions when a condition occurs. For example, when many models are being fitted, it may be desirable to continue fitting the others even if one fails to converge.

We conclude with a discussion of "defensive" programming and suggest strategies to avoid common errors before they occur. Quite often this involves spending a bit more time when writing the code but this pays off in the long run by reducing debugging time substantially. The basic principle of defensive programming is to raise an error as soon as something goes wrong. This takes three particular forms: checking that inputs are correct, avoiding non-standard evaluation, and avoiding functions that can return different types of output.

Debugging Techniques

Debugging code is challenging. Most bugs are subtle and hard to find otherwise we would probably have avoided them in the first place. Here, we discuss some useful tools, provided by R and RStudio, and outline a general procedure for debugging. While the procedure below is by no means foolproof, it is helpful in organizing your thoughts when debugging. There are five steps:

1. **Realize that you have a bug**: You cannot fix a bug until you know it exists. Hence, it is always a good idea to test your code before you finish. Automated testing is a more involved option in that direction.
2. **Make it repeatable**: Once you have determined you have a bug in your code, you need to be able to reproduce it on demand. Without this, it becomes extremely difficult to isolate its cause and to confirm that you have successfully fixed it. Generally, you will start with a big block of code that you know causes the error and then slowly whittle it down to get to the smallest possible snippet that causes the error. Binary search is particularly useful for this. To do a binary search, you repeatedly remove half of the code until you find the bug. This is fast because, with each step, you reduce the amount of code to look through by half. As you work on creating a minimal example, you will also discover similar inputs that do not trigger the bug. Make sure you take note of them as they will be helpful when diagnosing the cause of the bug.
3. **Figure out where it is**: If you are lucky, one of the tools in the following section will help you to quickly identify the line of code that is causing the bug. Usually, however, you will have to think a bit more about the problem. It is a good idea to adopt the scientific method. Generate hypotheses, design experiments to test them, and record your results. This may seem like a lot of work, but a systematic approach will end up saving you time.
4. **Fix it and test it**: Once you have found the bug, you need to figure out how to fix it and to check that the fix actually worked. Again, it is very useful to have automated tests in place. Not only does this help to ensure that you have actually fixed the bug, it also helps to ensure you have not introduced any new bugs in the process. In the absence of automated tests, make sure to carefully record the correct output, and check against the inputs that previously failed.
5. **Internet search**: In some cases, it may be a good idea to do an internet search on the error message. There is a chance that this is a common error with a known solution.

Debugging Tools

There are three key debugging tools in R:

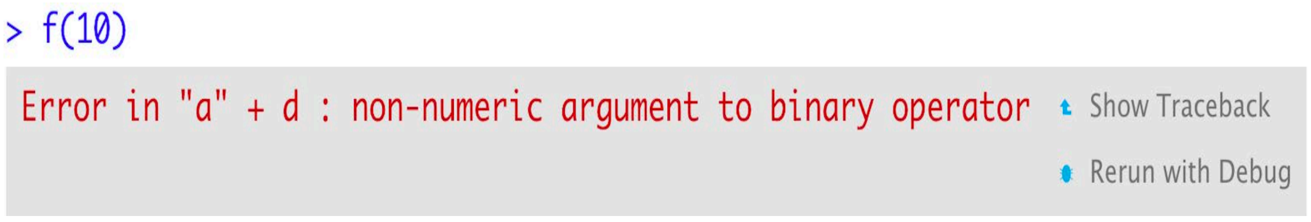
- RStudio's error inspector and `traceback()`, which list the sequence of calls that lead to the error.
- RStudio's "Rerun with Debug" tool and `options(error = browser)`, which open an interactive session where the error occurred.
- RStudio's breakpoints and the `browser()` function, which open an interactive session at an arbitrary location in the code.

Determining the Sequence of Calls

The first tool is the call stack, the sequence of calls that lead up to an error. Let us look at the following example: you can see that `f()` calls `g()` calls `h()` calls `i()`, which adds together a number and a string creating a error:

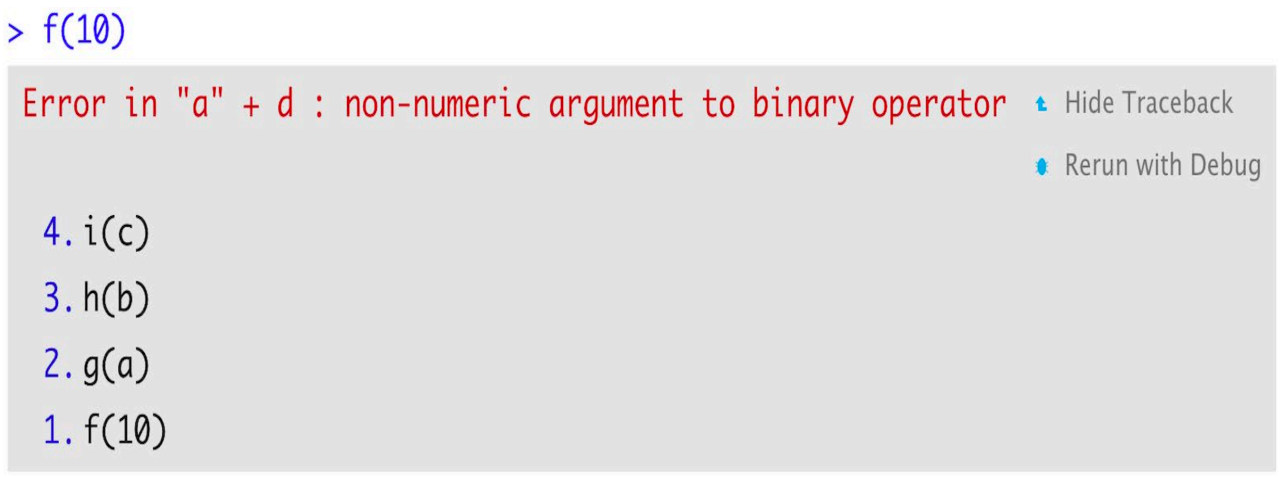
```
f <- function(a) g(a)
g <- function(b) h(b)
h <- function(c) i(c)
i <- function(d) "a" + d
f(10)
```

When we run this code in RStudio we see:



```
> f(10)
Error in "a" + d : non-numeric argument to binary operator
```

Two options appear to the right of the error message: "Show Traceback" and "Rerun with Debug." If you click "Show Traceback" you see:



```
> f(10)
Error in "a" + d : non-numeric argument to binary operator
  4. i(c)
  3. h(b)
  2. g(a)
  1. f(10)
```

Sometimes this is enough information to let you track down the error and fix it. However, sometimes it is not. `traceback()` shows where the error occurred, but not why. The next useful tool is the interactive debugger, which allows you to pause execution of a function and interactively explore its state.

Browsing on Error

The easiest way to enter the interactive debugger is through RStudio's "Rerun with Debug" tool. This reruns the command that created the error, pausing execution where the error occurred. You are now in an interactive state inside the function, and you can interact with any object defined there. You will see the corresponding code in the editor (with the statement that will be run next highlighted), objects in the current environment in the "Environment" pane, the call stack in a "Traceback" pane, and you can run arbitrary R code in the console. There are a few special commands you can use in debug mode. You can access them either with the RStudio toolbar or with the keyboard:

- *Next*: Executes the next step in the function. Be careful if you have a variable named `n`; to print it you will need to do `print(n)`.
- *Step into*: Works like Next, but if the next step is a function, it will step into that function so you can work through each of that functions lines.
- *Finish*: Finishes execution of the current loop or function.
- *Continue*: Leaves interactive debugging and continues regular execution of the function. This is useful if you have fixed the bad state and want to check that the function proceeds correctly.
- *Stop*: Stops debugging, terminates the function, and returns to the global workspace. Use this once you have figured out where the problem is, and you are ready to fix it and reload the code.

As well as entering an interactive console on error, you can enter it at an arbitrary code location by using either an RStudio breakpoint or `browser()`.

Condition (Error) Handling

Unexpected errors require interactive debugging to figure out what went wrong. Some errors, however, are expected, and you want to handle them automatically and in some cases ignore them. For example, when you are fitting a model to many datasets you may prefer to try and fit it into as many datasets as possible, even if some fits fail, and perform diagnostics in the end.

In R, there are three tools for handling conditions (including errors):

- `try()` gives you the ability to continue execution even when an error occurs.
- `tryCatch()` lets you specify handler functions that control what happens when a condition is signalled.
- `withCallingHandlers()` is a variant of `tryCatch()` that establishes local handlers, whereas `tryCatch()` registers exiting handlers.

Ignoring Errors with `try()`

`try()` allows execution to continue even after an error has occurred. For example, normally if you run a function that throws an error, it terminates immediately and does not return a value. The below code chunk

```
f1 <- function(x) {
  log(x)
  10
}
f1("x")
```

stops with the error message "Error in log(x): non-numeric argument to mathematical function." However, if you wrap the statement that creates the error in `try()`, the error message will be printed but execution will continue:

```
f2 <- function(x) {
  try(log(x))
  10
}
f2("a")
Error in log(x) : non-numeric argument to mathematical function
[1] 10
```

Handling Conditions with `tryCatch()`

`tryCatch()` is a general tool for handling conditions: in addition to errors, you can take different actions for warnings, messages, and interrupts. With `tryCatch()` you map conditions to handlers, which are named functions that are called with the condition as an input. If a condition is signalled, `tryCatch()` will call the first handler whose name matches one of the classes of the condition. A handler function can do anything, but typically it will either return a value or create a more informative error message. For example, the `show_condition()` function below sets up handlers that return the type of condition signalled:

```
show_condition <- function(code) {
  tryCatch(code,
    error = function(c) "error",
    warning = function(c) "warning",
    message = function(c) "message"
  )
}
show_condition(stop("!"))
[1] "error"
show_condition(warning("?!"))
[1] "warning"
show_condition(message("?"))
[1] "message"
# If no condition is captured, tryCatch returns the
# value of the input
show_condition(10)
[1] 10
```

As an example, consider a simple function that checks if an argument is an even number. You might write the following

```
is_even <- function(n) {
  n %% 2 == 0
}
is_even(768)
is_even("two")
```

You can see that providing a string causes this function to raise an error. You could imagine though that you want to use this function across a list of different data types, and you only want to know which elements of that list are even numbers. You might think to write the following:

```
is_even_error <- function(n) {
  tryCatch(n %% 2 == 0,
           error = function(e) {
             FALSE
           })
}
is_even_error(714)
[1] TRUE
is_even_error("eight")
[1] FALSE
```

Defensive Programming

Defensive programming is the strategy of making code fail in a well-defined manner even when something unexpected occurs. A key principle of defensive programming is to signal an error as soon as something wrong is discovered. In R, the "fail fast" principle is implemented in three ways:

- Be strict about what you accept. For example, if your function is not vectorized in its inputs, but uses functions that are, make sure to check that the inputs are scalars.
- Avoid functions that use non-standard evaluation, like `subset`, `transform`, and `with`. These functions save time when used interactively, but because they make assumptions to reduce typing, when they fail, they often fail with uninformative error messages.
- Avoid functions that return different types of output depending on their input. The two biggest offenders are `[` and `sapply()`.

Useful Links and Resources

- A prototype of a condition system for R: by Robert Gentleman and Luke Tierney
- Beyond exception handling: conditions and restarts: by Peter Seibel, translated from Lisp to R
- Automated Testing

References

Wickham, H. (2019). *Advanced R*. Taylor & Francis Inc. CRC Press Inc.