Received: 23 March 2022 | Accepted: 19 July 2022

DOI: 10.1111/2041-210X.13961

Methods in Ecology and Evolution BRITISH ECOLOGICAL SOCIETY

PERSPECTIVE

A call for clean code to effectively communicate science

Alessandro Filazzola[1,2] | CJ Lortie[3,4]

[1]Apex Resource Management Solutions, Ottawa, ON, Canada

[2]Centre for Urban Environments, University of Toronto Mississauga, Mississauga, ON, Canada

[3]Department of Biology, York University, Toronto, ON, Canada

[4]The National Center for Ecological Analysis and Synthesis, UCSB, Santa Barbara, CA, USA

Correspondence
Alessandro Filazzola
Email: alex.filazzola@utoronto.ca

Funding information
Center for Urban Environments and School of Cities, Grant/Award Number: Post-doctoral Fellowship; NSERC, Grant/Award Number: Discovery Grant; University of Toronto

Handling Editor: Laura Graham

Abstract

1. Effective coding is fundamental to the study of biology. Computation underpins most research, and reproducible science can be promoted through clean coding practices. Clean coding is crafting code design, syntax and nomenclature in a manner that maximizes the potential to communicate its intent with other scientists. However, computational biologists are not software engineers, and many of our coding practices have developed ad hoc without formal training, often creating difficult-to-read code for others. Hard-to-understand code can thus be limiting our efficiency and ability to communicate as scientists with one another.
2. The purpose of this paper is to provide a primer on some of the practices associated with crafting clean code by synthesizing a transformative text in software engineering along with recent articles on coding practices in computational biology. We review past recommendations to provide a series of best practices that transform coding into a human-accessible form of communication.
3. Three common themes shared in this synthesis are the following: (a) code has value and you are responsible for its organization to enable clear *communication*, (b) use a *formatting* style to guide writing code that is easily understandable and consistent and (c) apply *abstraction* to emphasize important elements and declutter.
4. While many of the provided practices and recommendations were developed with computational biologists in mind, we believe there is wider applicability to any biologist undertaking work in data management or statistical analyses. Clean code is thus a crucial step forward in resolving some of the crisis in reproducibility for science.

KEYWORDS
open science, principles, programming, replication, reproducibility, science communication, transparency

1 | INTRODUCTION

Coding is increasingly integrated into the biological sciences. There is the argument that all biology is computational biology, where contemporary research has long been dependent on computers (Markowetz, 2017; Wilson et al., 2017). Code and software can thus support many dimensions of scientific analysis in biology through data management and analysis (Hampton et al., 2013; Marx, 2013; Nussinov et al., 2015), but there is still a need to discuss and examine general practices. We use computation to manage and interact with the biology that we study to varying extents (Del Ser et al., 2019; Gardner, 2019; Oudeyer, 2018). For some studies, such as those that model species responses to climate change (Barton et al., 2019; e.g. Elith & Leathwick, 2009; Pollock et al., 2014), computation forms the entirety of the study. We propose here that computation used in science is a form of communication and evidence. This is similar to how an ecologist can share field notes, photographs or descriptions of methods in journals that focus on methods and approaches. However, as with other forms of communication, the ability to understand information is central to knowledge development. Biologists often spend considerable effort in writing clear, understandable methods sections for experiments, but we should consider applying that same to our computational methods.

There is a need for clean code that we define here as the code that is easily communicated to and understood by other humans (Martin, 2009; Wickham & Grolemund, 2016). The increasing use of code in ecology and evolution comes at a time when there is also a reproducibility crisis defined as the inability to replicate previous findings through experimentation or analysis (Baker, 2016; Kelly, 2019; Maxwell et al., 2015). Although there are limited incentives for greater reproducibility (Fraser et al., 2020; Kelly, 2019) and some question what is truly reproducible (e.g. Filazzola & Cahill Jr, 2021), having code and data that are computationally reproducible (i.e. Hampton et al., 2017) is within our agency as biologists. There have been increased calls to share code and data with published data descriptions and analytical manuscripts (Powers & Hampton, 2019; Ram, 2013). This coincides with an increasing percentage of biologists using scripting languages, such as *R* or *Python*, for their research projects (Bassi, 2007; Lai et al., 2019). Publishing one's code is a step forward, but the ability to understand another's script is open for improvement. While there is an initial cost of time involved with writing clean code, the person most likely to experience a benefit is a future version of ourselves, returning to the project months or even years later. This later benefit will often be time saved, not having to struggle to understand prior intentions. We propose that additional focus can be applied to producing clean code for the biological sciences.

2 | WHY CLEAN CODE?

The ability to understand code is just as important as its ability to complete a task. It is important to do both because society has seen the widespread adoption of microprocessors in devices from cars to fridges to light switches, where the incorrect placement of a comma can cost time, money or lives (Martin, 2018). In an extreme case, the dual crashes of Boeing 737 Max-8s that cost hundreds of lives were attributed to a failure in the MCAS software controlling the plane's pitch (Johnston & Harris, 2019; Travis, 2019). As computational biologists, our responsibility is equally important because our research can go on to inform decisions such as medical treatments, ecological restoration and agricultural practices (Dobrow et al., 2004; Olden et al., 2014; Swinton et al., 2007). Understandable code can decrease the likelihood of errors or bugs and will certainly increase the ability to find errors upon review. Clean code is thus linked to the integrity of our science.

Clean coding can increase accessibility because well-written code can let collaborators reuse shared scripts or review them to facilitate communicating analytical methods. We frequently share or obtain chunks of code from online sources, such as Stack Overflow (Gómez et al., 2013; Rosen & Shihab, 2016). The easier that chunk is to understand, the more useable and reliable it will be. There are also differences in coding backgrounds between team members with some favouring certain libraries or packages. Clean code can also communicate the overall purpose and analytical steps regardless of the programming language being used. Writing clean code can thus open the black box of analyses among collaborators, facilitating our ability to produce expedient and reproducible results. There is an opportunity to describe high-level rules from computer science disciplines (Tariq et al., 2020) that can increase readability for computational biologists.

3 | THE BASICS OF CLEAN CODE

Computer science has worked to address the challenge of producing 'clean code'. There have been significant efforts to create a set of best practices for crafting code. One notable example comes from Robert C. Martin, a software engineer who has published five books on best coding practices. The first text, Clean Code: A Handbook of Agile Software Craftsmanship (Martin, 2009) is an excellent starting point for computational biologists. Although the book is written for Java, most practices are broadly applicable to computational biologists. We describe three themes in the Clean Code that are relevant for improving the readability of code: communication, formatting and abstraction.

3.1 | Communication

Code is a form of communication. Clean code will communicate the tasks being conducted without the need for comments, methods or supplemental sections. Code comments are annotations within code for the reader, not the machine, that are meant to provide additional information or clarity to what is being executed. There is certainly a reliance on comments in computational biology to communicate

the intention of code and some packages have been designed to automate the process of annotation (e.g. Verde Arregoitia, 2022). Detailing code actions in a comment may start with good intentions, but later this can become problematic as there is a substantial cost in maintaining (or failing to maintain) these comments (Vogel, 2013). Updates to comments are sometimes neglected, and remnant comments remain that disagree with the code. Other comments include code that has been commented out, remaining as a zombie in the script out of fear that deletion will remove something important that was long forgotten. There are many instances when the information in comments could be better represented inside the code itself. Relying on the code to communicate is often the best approach by (a) renaming functions to describe their operations, (b) removing comments that repeat arguments, (c) removing commented-out code unless it is serving a purpose (Figure 1; Supplemental S1). The choice of names is also important, with style guides recommending functions be verbs and objects be nouns (e.g. function = comparePlantGrowth vs. object = plantGrowth) (Van Rossum et al., 2001; Wickham et al., 2019). Style guides for *Python* commonly recommended that objects with multiple items be pluralized (e.g. list = cars) (Van Rossum et al., 2001). Many computational biologists may aim for brevity in naming objects or functions, but a descriptive name will be more informative and will ultimately save time as it replaces the need for comments (e.g. Figure 1). Additionally, many integrative development environments (IDEs), such as VS *Code*, *RStudio or PyCharm*, support autocompletion reducing the need for short names. Designing the code itself to communicate information, and using comments sparingly for additional clarity, is often a direct way to impart information to other users.

The file structure and the order of code chunks can be as important to readability as the content of the code. More commonly referred to as software architecture, code organization includes creating an intuitive project design with predictable file structures, compartmentalizing functions, component separation and data management (Martin et al., 2018). This is a vast topic that goes beyond the scope of our manuscript. However, there are some aspects of code organization relevant to clean crafting in ecology and evolution. For instance, the use of header blocks or subheaders (i.e. sections of multi-lined comments) in code can be useful for identifying important breaks in the script. For a researcher, the header identifying the code chunks associated with specific predictions from their manuscript can facilitate connecting the analysis to the results. A header at the beginning of a script can also provide notes to the user for (re)use, warnings, required parameters or future development plants. Brevity should remain the goal and caution to avoid repeating anything that is already represented in the code. These headers are also not meant to be replacements for proper documentation, such as a formally deployed package, guide or vignette.

3.2 | Formatting

Coding requires many micro-decisions about formatting. These micro-decisions include the placement of whitespace, how to name

Original code chunk

```
## Create a function to run a linear model with plant growth compared in different groups

lmFun <- function(){
m1 <- lm(weight ~ group, ## response ~ predictor
           data = PlantGrowth) ## specify dataframe
  ## checked normality of response. Model = Normal
  ## shapiro.test(m1$residuals)
  anovaTable <- anova(m1) ## get ANOVA Table from model
  return(anovaTable)
}
lmFun()
```

Revised Chunk

```
lmComparePlantGrowth <- function(){
m1 <- lm(weight ~ group,
           data = PlantGrowth)
  anovaTable <- anova(m1)
  return(anovaTable)
}

lmComparePlantGrowth()
```

FIGURE 1 An example in *R* of frequently annotated code that can alternatively be conveyed within the code instead. In the revised chunk, many of the comments are removed since the functions capture the same information. The normality test that is commented out is removed, which can be returned if needed and currently serves no purpose. Lastly, the name of the function captures the purpose of the chunk and thus the first comment explaining the chunk is no longer necessary.

objects or functions and choice of functions, among many other formatting decisions. We provide nine formatting considerations when coding to help improve the cleanliness of code (Table 1). These formatting considerations can be generally summarized by providing adequate spacing, grouping like things together, and aiming for consistency. Spacing is a powerful tool in coding to separate chunks (paragraphs), lines (sentences) or arguments/objects (words), like the written word in most languages. Ideally, whitespace across lines, chunks, operators and words should be used strategically to indicate related and unrelatedness (dos Santos & Gerosa, 2018). In a sample script separating comma separate files, we demonstrate that spacing can improve the ease with which arguments are identified and our ability to understand the actions (Figure 2; Supplemental S1). Some individuals will advocate for certain formatting styles over others. For example, *tidyverse* in *R* recommends the use of snake_case as a naming convention (Wickham et al., 2019). We argue the best practices to use are the ones shared by the group you are working with.

There are tools available to support the consistent formatting of code. Standardized style guides can be an effective strategy for guiding code structure, including when to use code chunks, pipe operators and functions (Wickham, 2019). For those looking to take advantage of existing style guides, there are popular examples in many programming languages that can be modified, such as from Google or Mozilla (google.github.io/styleguide/, firefox-source-docs.mozilla.org/code-quality/coding-style/index.html). A similar guide already exists in ecology and evolution produced by the British Ecological Society (e.g. Cooper, 2017). There are popular extensions available in IDEs that can assist with real-time coding to produce clean code (Saini & Mussbacher, 2021). For example, *RStudio* has recently adopted a vertical line in the source code editor to indicate an 80-character line limit. Code formatters and linters can be used to enforce consistency by flagging stylistic errors or bugs. *Prettier* and Intellisense-based extensions in VS *Code* are popular among many programing languages that allow custom inputs for stylistic choices including formatting files on save, decisions on whitespace usage or returns after loops and functions (often called *linters*). Similar tools also exist for *R* (e.g. https://lintr.r-lib.org/, https://styler.r-lib.org/), Python (https://black.readthedocs.io/, flake8.pycqa.org) and Julia (https://domluna.github.io/JuliaFormatter.jl). In addition, there are other extensions that support auto-completion of entire chunks of code including 'snippet' libraries or Github Co-pilot (https://github.com/features/copilot/), which can support consistency between scripts and even developers. It is worth noting that these tools are not complete replacements for understanding appropriate

TABLE 1 Common best practices and guidelines for formatting code in biology. Some of these practices are language dependent and must be adjusted accordingly

Formatting	Description	Best practices	Examples
1. Horizontal length	The number of characters within a line	Between 80 and 120 characters with longer sections separated by returns	Functions with many arguments can have a return after each argument
2. Horizontal ordering	The order in which functions are executed	Nonmanipulative operations should occur before others	(1) Data loaded (2) Columns added (3) Aggregation
3. Horizontal spacing	The distance between characters separated by spaces	Space should be added between objects, arguments and operators	data = datasetName list = c(Obj1, Obj2)
4. Indenting	A starting position away from the margin	Indenting indicates nestedness of lines within a larger element	Used in functions, if/else statements and loops
5. Naming conventions	The text pattern for naming objects, functions or classes	Consistency is paramount. Consider different patterns for objects versus functions	snake_case camelCase Kebab-Case
6. Parentheses	Different bracket types to indicate closure around operations	Add horizontal or vertical spacing when appropriate to make obvious open/close parentheses. Try to minimize nesting parentheses	() [] {}
7. Vertical length	The total number of lines associated with a script	Script length is case-by-case but ideally less is more	A 1,000-line script could be split into multiple scripts (e.g. functions vs. analysis)
8. Vertical ordering	The order in which lines are presented in the script	Dependencies should be listed before dependents	(1) Libraries (2) Functions (3) Loading data (4) Analysis
9. Vertical spacing	The distance between lines separated by returns (i.e. lines)	Similar things should be grouped to indicate relatedness. The more lines between, the more unrelated	A linear model with all its diagnostic functions would be listed together

FIGURE 2 A *Python* script using nested for loops to separate species from the main tab-separated file into individual comma-separated files. In the original script, there is no use of white space to delineate similar items, shorten lines or separate code chunks. The revised script ends up being longer, but the arguments are more apparent.

Original Script

```
headerCols = ["gbifID","datasetKey","occurrenceID","kingdom","phylum","class","species","latitude","longitude"]

for i in csvFiles:
    iter_csv = pd.read_csv(i, sep="\t",usecols=keepCols,dtype=str,names=headerCols,header=0)
    uniqueSpp = list(set(iter_csv["species"]))
    for j in uniqueSpp:
        tempSpp=iter_csv[iter_csv["species"]==j]
        out_csv='~/project/data/speciesOcc/'+str(j)+'.csv'
        if not os.path.isfile(out_csv):
            tempSpp.to_csv(out_csv, index=False,header=keepCols,mode='w')
        else:
            tempSpp.to_csv(out_csv, index=False, header=False,mode='a')
```

Revised Script

```
headerCols = ["gbifID", "datasetKey", "occurrenceID", "kingdom", "phylum",
  "class", "species", "latitude", "longitude"]

for i in csvFiles:
    iter_csv = pd.read_csv(i,
      sep = "\t",
      usecols = keepCols,
      dtype = str,
      names = headerCols,
      header = 0)

    uniqueSpp = list(set(iter_csv["species"]))

    for j in uniqueSpp:
        tempSpp = iter_csv[iter_csv["species"] == j]

        out_csv = '~/project/data/speciesOcc/' + str(j) + '.csv'

        if not os.path.isfile(out_csv):
            tempSpp.to_csv(out_csv,
              index = False,
              header = keepCols,
              mode = 'w')
        else:
            tempSpp.to_csv(out_csv,
              index = False,
              header = False,
              mode = 'a')
```

formatting, but rather support crafting code by reducing tedium and increasing time spent reviewing or refactoring (i.e. improving code cleanliness). Style guides establish a coherent primer for well-formatted code and there is a suite of tools available to allow easy implementation and adherence to formatting rules.

Another element of formatting is the vertical and horizontal ordering of code (Table 1). Vertical ordering includes how lines are presented in the script and often has a formulaic approach. Computational biologists will often approach this formula sequentially with (a) loading libraries/packages, (b) specifying user-defined functions, (c) loading datasets, (d) data management or manipulation and (e) analysis. Code should be presented in a linear manner where downstream processes depend on upstream actions. To assist in tracking dependencies within projects and help create succinct scripts, users can turn to workflow management packages such as *targets* or *scipiper* in *R* (Appling, 2020; Landau, 2021), *SnakeMake* in *Python* (Mölder et al., 2021) and *GNU-Make* in *Unix* (Baker, 2020). Horizontal ordering is crucial for pipe-style function chaining. Restricting lines to 80–120 characters is recommended (Martin, 2009) and building code through incremental layers not only improves the ability to understand code but also allows opportunities to break lines (Wilkinson, 2012). It is important to note that order matters and that functions which add rows or columns should be performed before functions that collapse or summarize. Clean code typically also omits argument names within functions that are common or routine (i.e. defaults are set to the most common parameters). These recommended best practices keep code tidy through consistent ordering, a readable flow of scientific reasoning (Grolemund, 2014), a limited need for commenting rationales and mechanisms to reduce repetitive lines of code. These salient principles support high-level heuristics for literate coding in many programming languages.

3.3 | Abstraction

Abstraction in computer programming involves removing complex elements to increase focus and emphasis on what is important to the user (Shaw, 1984). In computational biology, this would involve extracting a series of functions from a single code chunk to make obvious what is occurring. To better describe abstraction, we present three code chunks of data preparation before running a logistic regression (Figures 3 and 4; Supplemental S1). In this example, the underlying code is identical across the three code chunks, where the observations of a species are loaded to extract climate values, cleaned to remove duplicates or missing values, checked for collinearity issues, and then a logistic regression is conducted (Figure 3). The difference among these chunks is the presentation of the code, where specific operations are wrapped into functions that are loaded elsewhere in the script. While often this can increase the quantity of

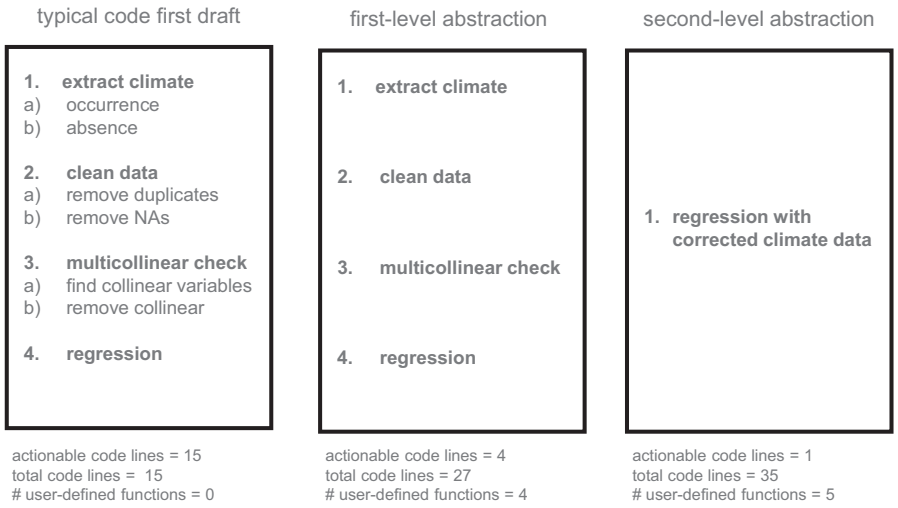

FIGURE 3 An outline of the abstraction steps taken in Figure 4 to replace chunks of code with descriptive functions. Lines of code can be grouped based on similar actions and then nested into functions. Abstraction will typically increase the total lines of code overall because additional lines are needed to include the syntax for user-defined functions. However, the number of actional code items that require user inputs or edits will decrease substantially.

code written, it certainly can increase the focus of the reader on tasks that are being executed. For future reviewers or users of this script, the most abstracted example makes abundantly clear the purpose of this code and provides brevity in the actionable lines of code. We recognize this is likely one of the more controversial topics in cleaning code. As scientists, we strive for full transparency and reproducibility (Baker, 2016; Fraser et al., 2020). Abstraction may be viewed as a strategy to obfuscate the underlying operations of code by packaging up details in functions that exist elsewhere in the script or another script entirely. Alternately, when used inappropriately, abstraction can lead to unnecessary complexity. Abstraction should break down complex code chunks into smaller, self-explanatory tasks to better describe the purpose or the script. Chunks of code that share similar roles, such as the data cleaning steps in Figure 4, can be nested into a single function, and those functions nested into a larger function (i.e. higher-order function) that has a more complex outcome (e.g. conduct a linear model). We argue abstraction significantly increases the cleanliness of the code and that those details still exist when needed for someone reviewing the project.

Applying abstraction is as much of an art as it is a science. The coding languages we often use in computational biology already have an inherent level of abstraction being high-level dynamic languages (e.g. *Julia*, *R*, *Python*). Consequently, it can be difficult to determine when abstraction should be applied to create a new function or dedicated class. We recommend the following three elements when applying abstraction to a project: map, stepdown and simplify. First, it is important to map an entire script, similar to outlining a manuscript before writing the text. No specifics are needed at this stage, simply the identification of code chunks that will answer project objectives. Second, with the completed outline we can apply the stepdown rule so that high-level functions are run first with low-level functions executed within the function (Martin, 2009). This approach does require some practice to learn because involves treating actions within the script as going from high-to-low levels rather than sequentially from top-to-bottom. Lastly, one should aim to simplify certain chunks as much as possible. Simplification may be accomplished by bundling integrated processes into functions (i.e. Figure 3), but this process can also be achieved by transforming repetitive steps into a function. However, it is notable that this effort at improving clarity can, at times, come at a cost to efficiency. For example, while *for loops* tend to be difficult to read relative to *map* functions, the loops can be faster (see Crites, 2018). Using these three steps (map, stepdown and simplify), we can apply abstraction to increase the readability of our code.

4 | PRINCIPLES OF CLEAN CODING

Computer science has developed a suite of best practices to use when crafting code each with varying applicability for the computational biologist. Some are more consistently relevant that are worth mentioning for further consideration by readers when crafting code. For instance, the DRY principle (do not repeat yourself) is an important axiom to minimize code repetition that can certainly facilitate readability (Thomas & Hunt, 2019). There are also the multiple principles that have been popularized in object-oriented and functional coding languages. For example, the single-responsibility principle, where a function should only conduct a single task (Martin et al., 2003). In our provided example (Figure 3) wrapping the entire original chunk into a single function would violate this principle without first breaking each action into its own set of functions (i.e. the first level of abstraction). This approach

FIGURE 4 Three levels of abstraction to conduct a logistic regression using climate data extracted from survey locations of a target species. In the first level of abstraction, sections of the code chunk are wrapped into functions to improve clarity. For instance, the data cleaning steps are wrapped into a data cleaning function. In the second level of abstraction, one can wrap the entire chunk into a single function (*runClimateLogisticRegression*) offering the reader the opportunity to focus more on the other elements of data preparation for the subsequent analysis, data predictions or visualizations.

Data load

```
library(terra)
library(usdm)

## Data load
speciesOccurrences <- vect("occurrence_points.shp")
speciesAbsences <- vect("absence_points.shp")
climateData <- rast("./climateData.tif")
```

Original code

```
## Climate extraction
occurenceClimate <- extract(climateData, speciesOccurrences)
absenceClimate <- extract(climateData, speciesAbsences)
allClimate <- rbind(occurenceClimate, absenceClimate)
allClimate[,"speciesRecord"] <- c(rep(1, nrow(occurenceClimate)),rep(0, nrow(occurenceClimate)))

## Data cleaning
allClimateNAremoved <- allClimate[complete.cases(allClimate),]
allClimateNADupsremoved <- allClimateNAremoved[!duplicated(allClimateNAremoved),]

## Check for multicollinearity
collinear <- vifcor(allClimateNADupsremoved[-ncol(allClimateNADupsremoved)])
removeVariables <-  collinear@excluded
allClimateModelReady <-  allClimateNADupsremoved[!names(allClimateNADupsremoved) %in% removeVariables]

## Run logistic regression
modelOut <- glm(speciesRecord ~ ., data = allClimateModelReady, family = binomial)
summary(modelOut)
```

First level of abstraction

```
extractClimateFromPoints <- function(occ, abs, climate) {
occurenceClimate <- extract(climate, occ)
absenceClimate <- extract(climate, abs)
climateDF <- rbind(occurenceClimate, absenceClimate)
climateDF[,"speciesRecord"] <- c(rep(1, nrow(occurenceClimate)),rep(0, nrow(occurenceClimate)))
return(climateDF)
}

cleanRecords <- function(climateDF) {
climateDF <- climateDF[complete.cases(climateDF),]
climateDF <- climateDF[!duplicated(climateDF),]
return(climateDF)
}

checkCollinear <- function(climateDF) {
collinear <- vifcor(climateDF[-ncol(climateDF)])
removeVariables <-  collinear@excluded
climateVariables <- climateDF[!names(climateDF) %in% removeVariables,]
return(climateVariables)
}

climateLogisticRegression <- function(processedClimateDF) {
model <- glm(speciesRecord ~ ., data = processedClimateDF, family = binomial)
print(summary(model))
return(model)
}

allClimate <- extractClimateFromPoints(speciesOccurrences, speciesAbsences, climateData)

allClimateNADupsremoved <- cleanRecords(allClimate)

allClimateModelReady <- checkCollinear(allClimateNADupsremoved)

modelOut <- climateLogisticRegression(allClimateModelReady)
```

Second level of abstraction

```
runClimateLogisticRegression <- function(occ, abs, climate) {
allClimate <- extractClimateFromPoints(speciesOccurrences, speciesAbsences, climateData)
allClimateNADupsremoved <- cleanRecords(allClimate)
allClimateModelReady <- checkCollinear(allClimateNADupsremoved)
climateLogisticRegression(allClimateModelReady)
return(climateLogisticRegression)
}

runClimateLogisticRegression(speciesOccurrences, speciesAbsences, climateData)
```

improves readability and minimizes obscurity by keeping functions simple to understand. Jenny Bryan in a UseR plenary provided an excellent set of tips to improve the way '*Code smells and feels*', such as minimizing indentation, revising nested if-else statements into separate functions when possible, and avoiding complicated Boolean expressions (Bryan, 2018).

Team composition is an important factor in determining the readability of code. The structure, diversity of experience and skill set of team members will lead to different adopted practices. The best formatting style guide is the one shared by team members, but there are other challenges for team-based coding. One significant challenge can be relative differences in the skill set. Relatively more novice individuals may focus more on immediate functionality rather than higher-level repeatability or readability. In computer science, there is the expression 'make it work, make it right' (Beck, 2003) that certainly also applies to computation in ecology and evolution. First, get the code working. Second, spend the time improving the code as a team to ensure repeatability and readability by others. Different team members, for instance, initially comment more often to provide explanations for what arguments certain functions require, eventually transitioning to removing these comments. The technique for improving the design of preexisting code is called *refactoring*, which is a common practice in computer science (Fowler, 2018). Any return visits to code should likely include design improvements, regardless of how small the change, to improve future use and inevitably lead to substantial change (Fowler, 2018). Clean code is thus not a binary (i.e. clean vs. not clean) designation, especially in collaborator work. Instead, clean code is a continuum that includes flexibility in shared practices adopted by the team, so as not to exclude junior members and respect the differences in the need to annotate functions or steps.

5 | IMPLICATIONS

Consistency between the structure and semantics of data can provide benefits beyond legibility, particularly facilitating clean thinking in project design. The incredible flexibility of interpreted programming languages is that there are often many solutions to a single challenge including data munging (Dasu & Johnson, 2003). Nonetheless, a set of practices and preferred formatting reduce the cognitive costs of mentally manipulating representations from data structures to code to more extended workflows including functions and code chunks or modules (Menary & Kirchhoff, 2014). Applications of clean code can thus result in relatively consistent formats thereby lowering the mental overhead for computational biologists (Al-Fedaghi, 2021; Codd, 1990). This seamlessness can be further enhanced by style guides or generalized cultural practices within a community of programmers, although these practices vary between groups. Some packages and libraries ensure code conforms to the style guides and enables modularized code reuse. Adopting style guides increases consistency in coding through shared styles that are clean and/or tidy for instance lowers the number of decisions needed for the programmers that are not directly associated with the purpose of the work and further reduces the likelihood of mistakes.

We recognize that clean code is not a silver bullet in solving computational reproducibility, but rather one of many skills for improving the transparency of our science. For example, avoiding complexity in software or analytics through subdivision, documentation and review can make our code inherently more understandable (Vedder et al., 2021). The increasing popularity and awareness of the '*10 Simple Rules*' series also provides multiple great recommendations for improving computational reproducibility such as using version control, providing ancillary documentation (e.g. README files, worked examples, vignettes) and automation (Lee, 2018; Sandve et al., 2013; Wilson et al., 2017). Version control, such as with Git, can be especially powerful in providing a history of code changes, reducing the need to preserve old or unused code. However, the benefits from each of these recommendations would be best realized when writing clean code. While most biologists did not get into the discipline to become computer programmers, there are certainly some 'good enough' practices to enable reuse, replication and understanding (Wilson et al., 2017). Among the list of good enough practices is the common theme of sharing code and collaboration (Wilson et al., 2017), which can only be achieved if the code is understandable.

We synthesized recommendations from *Clean Code* (Martin, 2009) along with general recommendations presented in scripting languages such as *R* and *Python* (Van Rossum et al., 2001; Wickham et al., 2019). There are certainly many fundamental principles and common decisions to be shared among computational biologists. In our synthesis, we offer the following three recommendations: (a) code has value and you are responsible for its organization to enable clear *communication*, (b) use a *formatting* style to guide writing code that is easily understandable and consistent and (c) apply *abstraction* to emphasize important elements and declutter. We advocate these recommendations are applied at project conception to facilitate project success and encourage the publication of code. Clean code improves computational reproducibility, a requirement for many journals, but also a significant benefit for our future selves. There is also a broader societal benefit as there is increased transparency and computational reproducibility among biologists.

AUTHOR CONTRIBUTIONS

Both authors contributed equally to the conception, design and example creation. Alessandro Filazzola led the writing of the manuscript. Both authors contributed critically to the drafts and gave final approval for publication.

ACKNOWLEDGEMENTS

We thank Sophie Breitbart, Matthew R. Brousil, James Cahill Jr., Michael F. Meyer, Courtney Robichaud and two anonymous reviewers for their comments on an earlier draft and suggestions for general coding practices. We also thank Robert Martin for his comments relating clean code to the R Users MeetUp group. That presentation and discussion inspired this perspective. This research was

funded by a postdoctoral fellowship awarded to A.F. by the Center for Urban Environments and School of Cities at the University of Toronto, Canada, and an NSERC DG to C.J.L.

CONFLICT OF INTEREST

None of the authors have a conflict of interest to declare.

PEER REVIEW

The peer review history for this article is available at https://publons.com/publon/10.1111/2041-210X.13961.

DATA AVAILABILITY STATEMENT

No data were associated with this manuscript. Code snippets can be found online at the following repository https://doi.org/10.5281/zenodo.6917279 (Filazzola, 2022).

ORCID

Alessandro Filazzola https://orcid.org/0000-0001-6544-2035
CJ Lortie https://orcid.org/0000-0002-4291-7023

REFERENCES

Al-Fedaghi, S. (2021). *Conceptual temporal modeling applied to databases*.

Appling, A. (2020). scipiper: Support functions for ushering data through a scientific workflow. R package version 0.0.24. Retrieved from https://github.com/USGS-R/scipiper

Baker, M. (2016). 1,500 scientists lift the lid on reproducibility. *Nature*, *533*, 452–454.

Baker, P. (2020). Using GNU make to manage the workflow of data analysis projects. *Journal of Statistical Software*, *94*, 1–46.

Barton, M. G., Terblanche, J. S., & Sinclair, B. J. (2019). Incorporating temperature and precipitation extremes into process-based models of African lepidoptera changes the predicted distribution under climate change. *Ecological Modelling*, *394*, 53–65. https://doi.org/10.1016/j.ecolmodel.2018.12.017

Bassi, S. (2007). A primer on python for life science researchers. *PLoS Computational Biology*, *3*(11), e199. https://doi.org/10.1371/journal.pcbi.0030199

Beck, K. (2003). *Test-driven development: By example*. Addison-Wesley Professional.

Bryan, J. (2018). *Code smells and feels*. UseR: The conference for users of R. Retrieved from https://www.youtube.com/watch?v=7oyiPBjLAWY&t=2295s&ab_channel=RConsortium

Codd, E. F. (1990). *The relational model for database management: Version 2*. Addison-Wesley Longman Publishing Co., Inc.

Cooper, N. (2017). *A guide to reproducible code in ecology and evolution*. British Ecological Society.

Crites, A. (2018). *map vs. for loop*. Medium. Retrieved from https://medium.com/@ExplosionPills/map-vs-for-loop-2b4ce659fb03

Dasu, T., & Johnson, T. (2003). *Exploratory data mining and data cleaning*. John Wiley & Sons.

Del Ser, J., Osaba, E., Molina, D., Yang, X.-S., Salcedo-Sanz, S., Camacho, D., Das, S., Suganthan, P. N., Coello Coello, C. A., & Herrera, F. (2019). Bio-inspired computation: Where we stand and what's next. *Swarm and Evolutionary Computation*, *48*, 220–250. https://doi.org/10.1016/j.swevo.2019.04.008

Dobrow, M. J., Goel, V., & Upshur, R. E. G. (2004). Evidence-based health policy: Context and utilisation. *Social Science & Medicine*, *58*(1), 207–217. https://doi.org/10.1016/S0277-9536(03)00166-7

dos Santos, R. M., & Gerosa, M. A. (2018). Impacts of coding practices on readability. In *International Conference on Software Engineering. Proceedings of the 26th Conference on Program Comprehension* (pp. 277–285). Association for Computing Machinery.

Elith, J., & Leathwick, J. R. (2009). Species distribution models: Ecological explanation and prediction across space and time. *Annual Review of Ecology, Evolution, and Systematics*, *40*(1), 677–697. https://doi.org/10.1146/annurev.ecolsys.110308.120159

Filazzola, A. (2022). afilazzola/CleanCodeEvoEco: OriginalRelease (1.0). *Zenodo*, https://doi.org/10.5281/zenodo.6917279

Filazzola, A., & Cahill, J. F., Jr. (2021). Replication in field ecology: Identifying challenges and proposing solutions. *Methods in Ecology and Evolution*, *12*(10), 1780–1792. https://doi.org/10.1111/2041-210X.13657

Fowler, M. (2018). *Refactoring: Improving the design of existing code*. Addison-Wesley Professional.

Fraser, H., Barnett, A., Parker, T. H., & Fidler, F. (2020). The role of replication studies in ecology. *Ecology and Evolution*, *10*(12), 5197–5207. https://doi.org/10.1002/ece3.6330

Gardner, J. L. (2019). Optimality and heuristics in perceptual neuroscience. *Nature Neuroscience*, *22*(4), 514–523. https://doi.org/10.1038/s41593-019-0340-4

Gómez, C., Cleary, B., & Singer, L. (2013). A study of innovation diffusion through link sharing on stack overflow. *2013 10th working conference on mining software repositories (MSR)*, 81–84. https://doi.org/10.1109/MSR.2013.6624011

Grolemund, G. (2014). *Hands-on programming with R: Write your own functions and simulations*. O'Reilly Media, Inc.

Hampton, S. E., Jones, M. B., Wasser, L. A., Schildhauer, M. P., Supp, S. R., Brun, J., Hernandez, R. R., Boettiger, C., Collins, S. L., Gross, L. J., Fernández, D. S., Budden, A., White, E. P., Teal, T. K., Labou, S. G., & Aukema, J. E. (2017). Skills and knowledge for data-intensive environmental research. *Bioscience*, *67*(6), 546–557.

Hampton, S. E., Strasser, C. A., Tewksbury, J. J., Gram, W. K., Budden, A. E., Batcheller, A. L., Duke, C. S., & Porter, J. H. (2013). Big data and the future of ecology. *Frontiers in Ecology and the Environment*, *11*(3), 156–162. https://doi.org/10.1890/120103

Johnston, P., & Harris, R. (2019). The Boeing 737 MAX saga: Lessons for software organizations. *Software Quality Professional*, *21*(3), 4–12.

Kelly, C. D. (2019). Rate and success of study replication in ecology and evolution. *PeerJ*, *7*, e7654.

Lai, J., Lortie, C. J., Muenchen, R. A., Yang, J., & Ma, K. (2019). Evaluating the popularity of R in ecology. *Ecosphere*, *10*(1), e02567. https://doi.org/10.1002/ecs2.2567

Landau, W. M. (2021). The targets R package: A dynamic make-like function-oriented pipeline toolkit for reproducibility and high-performance computing. *Journal of Open Source Software*, *6*(57), 2959.

Lee, B. D. (2018). Ten simple rules for documenting scientific software. *PLoS Computational Biology*, *14*(12), e1006561. https://doi.org/10.1371/journal.pcbi.1006561

Markowetz, F. (2017). All biology is computational biology. *PLoS Biology*, *15*(3), e2002050. https://doi.org/10.1371/journal.pbio.2002050

Martin, R. (2018). *Rabobank & Utrecht JUG present: Clean coding with uncle bob*. Utrecht Java User Group. Retrieved from https://www.meetup.com/Utrecht-Java-User-Group/events/257794733/?comment_table_id=260633274&comment_table_name=reply

Martin, R. C. (2009). *Clean code: A handbook of agile software craftsmanship*. Pearson Education.

Martin, R. C., Grenning, J., Brown, S., Henney, K., & Gorman, J. (2018). *Clean architecture: A craftsman's guide to software structure and design*. Prentice Hall.

Martin, R. C., Newkirk, J., & Koss, R. S. (2003). *Agile software development: Principles, patterns, and practices* (Vol. 2). Prentice Hall.

Marx, V. (2013). The big challenges of big data. *Nature*, *498*(7453), 255–260. https://doi.org/10.1038/498255a

Maxwell, S. E., Lau, M. Y., & Howard, G. S. (2015). Is psychology suffering from a replication crisis? What does "failure to replicate" really mean? *American Psychologist*, *70*(6), 487–498.

Menary, R., & Kirchhoff, M. (2014). Cognitive transformations and extended expertise. *Educational Philosophy and Theory*, *46*(6), 610–623.

Mölder, F., Jablonski, K. P., Letcher, B., Hall, M. B., Tomkins-Tinch, C. H., Sochat, V., Forster, J., Lee, S., Twardziok, S. O., Kanitz, A., Wilm, A., Holtgrewe, M., Rahmann, S., Nahnsen, S., & Köster, J. (2021). Sustainable data analysis with Snakemake. *F1000Research*, *10*, 33.

Nussinov, R., Bonhoeffer, S., Papin, J. A., & Sporns, O. (2015). From "what is?" to "what Isn't?" Computational Biology. *PLOS Computational Biology*, *11*(7), e1004318. https://doi.org/10.1371/journal.pcbi.1004318

Olden, J. D., Konrad, C. P., Melis, T. S., Kennard, M. J., Freeman, M. C., Mims, M. C., Bray, E. N., Gido, K. B., Hemphill, N. P., Lytle, D. A., McMullen, L. E., Pyron, M., Robinson, C. T., Schmidt, J. C., & Williams, J. G. (2014). Are large-scale flow experiments informing the science and management of freshwater ecosystems? *Frontiers in Ecology and the Environment*, *12*(3), 176–185. https://doi.org/10.1890/130076

Oudeyer, P.-Y. (2018). *Computational theories of curiosity-driven learning*.

Pollock, L. J., Tingley, R., Morris, W. K., Golding, N., O'Hara, R. B., Parris, K. M., Vesk, P. A., & McCarthy, M. A. (2014). Understanding co-occurrence by modelling species simultaneously with a joint species distribution model (JSDM). *Methods in Ecology and Evolution*, *5*(5), 397–406. https://doi.org/10.1111/2041-210X.12180

Powers, S. M., & Hampton, S. E. (2019). Open science, reproducibility, and transparency in ecology. *Ecological Applications*, *29*(1), e01822. https://doi.org/10.1002/eap.1822

Ram, K. (2013). Git can facilitate greater reproducibility and increased transparency in science. *Source Code for Biology and Medicine*, *8*(1), 7. https://doi.org/10.1186/1751-0473-8-7

Rosen, C., & Shihab, E. (2016). What are mobile developers asking about? A large scale study using stack overflow. *Empirical Software Engineering*, *21*(3), 1192–1223. https://doi.org/10.1007/s10664-015-9379-3

Saini, R., & Mussbacher, G. (2021). Towards conflict-free collaborative modelling using VS code extensions. *2021 ACM/IEEE international conference on model driven engineering languages and systems companion (MODELS-C)*, 35–44. https://doi.org/10.1109/MODELS-C53483.2021.00013

Sandve, G. K., Nekrutenko, A., Taylor, J., & Hovig, E. (2013). Ten simple rules for reproducible computational research. *PLoS Computational Biology*, *9*(10), e1003285. https://doi.org/10.1371/journal.pcbi.1003285

Shaw, M. (1984). Abstraction techniques in modern programming languages. *IEEE Software*, *1*(4), 10–26.

Swinton, S. M., Lupi, F., Robertson, G. P., & Hamilton, S. K. (2007). Ecosystem services and agriculture: Cultivating agricultural ecosystems for diverse benefits. *Ecological Economics*, *64*(2), 245–252. https://doi.org/10.1016/j.ecolecon.2007.09.020

Tariq, M. U., Bashir, M. B., Babar, M., & Sohail, A. (2020). Code readability management of high-level programming languages: A comparative study. *International Journal of Advanced Computer Science and Applications (IJACSA)*, *11*(3), 595–602.

Thomas, D., & Hunt, A. (2019). *The pragmatic programmer: Your journey to mastery*. Addison-Wesley Professional.

Travis, G. (2019). *How the Boeing 737 max disaster looks to a software developer*. IEEE Spectrum.

Van Rossum, G., Warsaw, B., & Coghlan, N. (2001). PEP 8—Style guide for python code. Python.Org.

Vedder, D., Ankenbrand, M., & Sarmento Cabral, J. (2021). Dealing with software complexity in individual-based models. *Methods in Ecology and Evolution*, *12*(12), 2324–2333. https://doi.org/10.1111/2041-210X.13716

Verde Arregoitia, L. D. (2022). *Annotater: Annotate package load calls*. https://github.com/luisDVA/annotater

Vogel, P. (2013). No comment: Why commenting code is still a bad idea. *Visual Studio Magazine*, 07/31. Retrieved from https://visualstudiomagazine.com/articles/2013/07/26/why-commenting-code-is-still-bad.aspx

Wickham, H. (2019). *Advanced r*. CRC Press.

Wickham, H., Averick, M., Bryan, J., Chang, W., McGowan, L. D., François, R., Grolemund, G., Hayes, A., Henry, L., Hester, J., Kuhn, M., Pedersen, T. L., Miller, E., Bache, S. M., Müller, K., Ooms, J., Robinson, D., Seidel, D. P., Spinu, V., … Yutani, H. (2019). Welcome to the Tidyverse. *Journal of Open Source Software*, *4*(43), 1686.

Wickham, H., & Grolemund, G. (2016). *R for data science: Import, tidy, transform, visualize, and model data*. O'Reilly Media, Inc.

Wilkinson, L. (2012). The grammar of graphics. In *Handbook of computational statistics* (pp. 375–414). Springer.

Wilson, G., Bryan, J., Cranston, K., Kitzes, J., Nederbragt, L., & Teal, T. K. (2017). Good enough practices in scientific computing. *PLoS Computational Biology*, *13*(6), e1005510. https://doi.org/10.1371/journal.pcbi.1005510

SUPPORTING INFORMATION

Additional supporting information can be found online in the Supporting Information section at the end of this article.

How to cite this article: Filazzola, A., & Lortie, C. (2022). A call for clean code to effectively communicate science. *Methods in Ecology and Evolution*, *00*, 1–10. https://doi.org/10.1111/2041-210X.13961