Good practices in bioinformatic or computational biology projects

Yang Cao

2021-12-25

Contents

About **5**
Project Organization 5
Code style 5

1 Project Organization **7**
1.1 README 7
1.2 Document 8
1.3 Data 8
1.4 Results 8
1.5 Code 8
1.6 Compiled programs 8
1.7 Example 8

Code Style **13**

2 Bash Shell Style Guide **13**
2.1 Comments 13
2.2 Naming conventions 14
2.3 Formatting 15
2.4 Quoting 16
2.5 Error Handing 17
2.6 Features 18
2.7 References 19
2.8 Resources 19

3 Python Style Guide **21**
3.1 Comments and Docstrings 21
3.2 Naming Conventions 25
3.3 Code Layout 26
3.4 References 32

4 R Style Guide **33**

4.1 Files 33
4.2 Naming Convention 34
4.3 Spacing 34
4.4 Function Calls 37
4.5 Control flow 40
4.6 Long lines 42
4.7 Semicolons 43
4.8 Assignment 43
4.9 Character vectors 43
4.10 Logical vectors 44
4.11 Comments 44
4.12 Resources 44

About

Welcome!

Good practices in bioinformatic or computational biology projects.

Bioinformatic or computational biology projects need to follow the same practices as lab projects and notebooks, with organized data, documented steps, the project structured for reproducibility, and the consistent code style.

Project Organization

It is generally a good idea to store all of the files relevant to one project under a common root directory. And organizing the files that make up a project in a logical and consistent directory structure will help you and others keep track of them.

The recommendations are drawn from:

Wilson, Greg, et al. "Good enough practices in scientific computing." PLoS computational biology 13.6 (2017): e1005510.

Code style

Write code in a consistent style will make it easier to maintain, and easier for collaborators to understand. Code is more often read than written, and code style can also have a great impact on the readability of your code

The style guides here are fundamentally opinionated, but not too opinionated. And many decisions are arbitrary, any suggestions are welcome.

The bash is derived from Google's shell style guide; python code style is derived from Google's python style guide and PEP 8 style guide for python; and the R code style is drived from Tidyverse Style Guide.

If you're modifying existing files, you should follow the style that's already in the files.

Do what Romans do in Rome.

Chapter 1

Project Organization

As a rule of thumb, divide work into projects based on the overlap in data and code files. If 2 research efforts share no data or code, they will probably be easiest to manage independently. If they share more than half of their data and code, they are probably best managed together, while if you are building tools that are used in several projects, the common code should probably be in a project of its own.

Projects do often require their own organizational model, but below are general recommendations on how you can structure data, code, analysis outputs, and other files. The important concept is that it is useful to organize the project by the types of files and that consistency helps you effectively find and use things later.

All files should be named using using `snake_case` to reflect their content or function.

1.1 README

`README` should be created in the root directory of the project to introduce and explain the project. It should at least cover the following terms:

- The project's title, a brief description.
- Dependencies and requirements, and how to install the requirements. If all the requirements have been installed on your server, please detailed that whether others should add the paths of the tools to the `PATH` variable. If a docker image which preinstalled all dependencies and requirements has been created, please provides details on how to use it.
- A simple example on how to run the analysis tasks of the project. an example or 2 of how to run various cleaning or analysis tasks.

- If the project is a software or pipeline, I recommend to write a detailed manual, which can help others to use it.

1.2 Document

Put text documents associated with the project in the `doc` directory. This includes files for manuscripts, documentation for source code, and/or an electronic lab notebook recording your experiments. Subdirectories may be created for these different classes of files in large projects.

1.3 Data

Put raw data and metadata in the `data` directory. The `data` directory might require subdirectories to organize raw data based on time, method of collection, or other metadata most relevant to your analysis.

1.4 Results

Files generated during cleanup and analysis in the `results` directory where "generated files" includes intermediate results such as cleaned data sets or simulated data, as well as final results such as figures and tables.

The `results` directory will usually require additional subdirectories. Intermediate files such as cleaned data, statistical tables, and figures should be separated clearly by file-naming conventions or placed into different subdirectories.

1.5 Code

`src` contains all of the code written for the project. This includes programs written in interpreted languages such as R or Python; those written in compiled languages like Fortran, C++, or Java; as well as shell scripts, snippets of SQL used to pull information from databases; and other code needed to regenerate the results.

1.6 Compiled programs

Compiled programs should be saved in the `bin` directory. Projects that do not have any executable programs compiled from code in the src directory will not require bin.

1.7 Example

- A `README` file that provides an overview of the project as a whole.

- The `data` directory contains the sequence file (machine-readable metadata could also be included here).
- The `src` directory contains `run_shapemap`, a Python file containing functions to analysis the shapemap data, `run_3wj` to perform 3WJ prediction, and a controller script `runall.py` that run all the analysis.
- Different results (shape and 3wj) are saved on their own subdirectories in the `results` directory.
- Optional: A `CITATION` file that explains how to reference it, and a `LICENSE` file that states the licensing.

```
|-- README

|-- requirements.txt

|-- data

|    |-- sample1.fq

|    |-- sample2.fq

|-- doc

|    |-- notebook.md

|    |-- manuscript.md

|    |-- changelog.txt

|-- results

|    |-- shapemap

|  |  |-- res.shapemap

|    |-- 3WJ

|  |  |-- 3wj.csv

|  |  |-- 4wj.csv

|  |  | -- ...

|-- src

|    |-- run_shapmap.py
```

```
|    |-- run_3wj.py
|    |-- runall.py
|-- CITATION
|-- LICENSE
```

Code Style

Chapter 2

Bash Shell Style Guide

2.1 Comments

2.1.1 File Header

Start each file with a description of its contents.

Every file must have a top-level comment including a brief overview of its contents. Author information are optional.

Example:

```
#!/bin/bash
#
# Author:
# Desc: Perform hot backups of Oracle databases.
```

2.1.2 Function Comments

Any function that is not both obvious and short must be commented. Any function in a library must be commented regardless of length or complexity.

It should be possible for someone else to learn how to use your program or to use a function in your library by reading the comments (and self-help, if provided) without reading the code.

All function comments should contain:

- Description of the function
- Global variables used and modified
- Arguments taken
- Returned values other than the default exit status of the last command run

Example:

```
#!/bin/bash
#
# Author:
# Desc: Cleanup files from the backup directory.
#######################################
# Globals:
#   BACKUP_DIR
#   ORACLE_SID
# Arguments:
#   None
#######################################
function cleanup() {
  …
}
```

2.1.3 Implementation Comments

Don't comment everything. Comment tricky, non-obvious, interesting or important parts of your code. use comments to explain the "why" not the "what" or "how". Each line of a comment should begin with the comment symbol and a single space: #.

2.2 Naming conventions

2.2.1 File Names

Lowercase, with underscores to separate words if desired. ‘

2.2.2 Function Names

Function names should be in snake_case. That is, all lower case and words are separated by underscores. Parentheses are required after the function name.

```
func() {
  ...
}

### Variable Names

As for function names. Variables names for loops should be similarly named for
any variable you’re looping through.

```sh
for zone in "${zones[@]}"; do
```
```

```
  something_with "${zone}"
done
```

2.2.3 Constants and Environment Variable Names

All caps, separated with underscores, declared at the top of the file.

2.3 Formatting

2.3.1 Identation

Indent 2 spaces. No tabs.

Use blank lines between blocks to improve readability. Indentation is two spaces. Whatever you do, don't use tabs. For existing files, stay faithful to the existing indentation.

2.3.2 Line Length

Maximum line length is 80 characters.

2.3.3 Pipe

Pipelines should be split one per line if they don't all fit on one line, and put pipe symbol (`|`) at the beginning of of its statement

If a pipeline all fits on one line, it should be on one line.

```
 # This is an inline pipe: "$(ls -la /foo/ | grep /bar/)"

 # The following pipe is of display form: every command is on
 # its own line

foobar="$( \
  ls -la /foo/ \
  | grep /bar/ \
  | awk '{print $NF}')"
```

2.3.4 Loops

Put `; do` and `; then` on the same line as the `while`, `for` or `if`. `else` should be on its own line and closing statements should be on their own line vertically aligned with the opening statement.

2.3.5 Variable Expansion

In order of precedence: Stay consistent with what you find; quote your variables; prefer `"${var}"` over `"$var"`.

They are listed in order of precedence.

- Stay consistent with what you find for existing code.
- Quote variables, see Quoting section below.
- Don't brace-delimit single character shell specials / positional parameters, unless strictly necessary or avoiding deep confusion.

```
# Section of *recommended* cases.

# Preferred style for 'special' variables:
echo "Positional: $1" "$5" "$3"
echo "Specials: !=$!, -=$-, _=$_. ?=$?, #=$# *=$* @=$@ \$=$$ …"

# Braces necessary:
echo "many parameters: ${10}"

# Braces avoiding confusion:
# Output is "a0b0c0"
set -- a b c
echo "${1}0${2}0${3}0"
```

Prefer brace-delimiting all other variables.

2.4 Quoting

- Always quote strings containing variables, command substitutions, spaces or shell meta characters.
- Optionally quote shell-internal, readonly special variables that are defined to be integers: `$?`, `$#`, `$$`, `$!`.
- Use double quotes for strings that require variable expansion or command substitution interpolation, and single quotes for all others.

```
# "Double" quotes indicate that substitution is required/tolerated.

# "quote variables"
echo "${flag}"

# double quote for strings that require variable expansion
bar="You are $USER"
# or command substitution
number="$(generate_number)"
```

```
# single quote for strings does not require variable expansion
foo='Hello World'

# "quote shell meta characters"
echo 'Hello stranger, and well met. Earn lots of $$$'
echo "Process $$: Done making \$\$\$."
```

2.5 Error Handing

All errors should be sent to `STDERR`.

2.5.1 Error Checking

`cd`, for example, doesn't always work. Make sure to check for any possible errors for `cd` (or commands like it) and exit or break if they are present.

```
# wrong
cd /some/path # this could fail
rm file       # if cd fails where am I? what am I deleting?

# right
cd /some/path || exit
```

2.5.2 set -e

Use `set -e` if your script is being used for your own business. Recommend do not use it.

```
# If _do_some_critical_check fails, the script just exits and the following
# code is just skipped without any notice.
set -e
_do_some_critical_check

if [[ $? -ge 1 ]]; then
  echo "Oh, you will never see this line."
fi
```

2.5.3 set -u

To make sure you won't use any undeclared variable , `set -u` is recommended.

2.6 Features

2.6.1 Command Substitution

Use `$(command)` instead of backticks.

```
# This is preferred
foo=`date`

# This is not
foo=$(date)
```

2.6.2 Math

Always use `(( … ))` or `$(( … ))` rather than `let` or `$[ … ]` or `expr`.

```
# Simple calculation used as text - note the use of $(( … )) within
# a string.
echo "$(( 2 + 2 )) is 4"

# When performing arithmetic comparisons for testing
if (( a < b )); then
  …
fi

# Some calculation assigned to a variable.
(( i = 10 * j + 400 ))
```

2.6.3 Listing Files

Listing Files Do not parse `ls(1)`, instead use bash builtin functions to loop files

```
# use
for f in *; do
    ...
done

# not
for f in $(ls); do
    ...
done
```

2.6.4 Arrays and lists

Use bash arrays instead of a string separated by spaces (or newlines, tabs, etc.) whenever possible.

```
# use array
modules=(json httpserver jshint)
for module in "${modules[@]}"; do
    npm install -g "$module"
done

# instead of string separated by spaces
modules='json httpserver jshint'
for module in $modules; do
    npm install -g "$module"
done
```

2.6.5 Test, [...], and [[...]]

`[[ … ]]` is preferred over `[ … ]`, `test`. `[[ … ]]` reduces errors as no pathname expansion or word splitting takes place between `[[` and `]]`. In addition, `[[ … ]]` allows for regular expression matching, while `[ … ]` does not.

```
# This ensures the string on the left is made up of characters in
# the alnum character class followed by the string name.
# Note that the RHS should not be quoted here.
if [[ "filename" =~ ^[[:alnum:]]+name ]]; then
  echo "Match"
fi

# This matches the exact pattern "f*" (Does not match in this case)
if [[ "filename" == "f*" ]]; then
  echo "Match"
fi
# This gives a "too many arguments" error as f* is expanded to the
# contents of the current directory
if [ "filename" == f* ]; then
  echo "Match"
fi
```

2.7 References

- Google shell style guide
- bash style guide
- bash coding style

2.8 Resources

Shellcheck

Chapter 3

Python Style Guide

3.1 Comments and Docstrings

You should use comments to document code as it's written. It is important to document your code so that you, and any collaborators, can understand it. When you or someone else reads a comment, they should be able to easily understand the code the comment applies to and how it fits in with the rest of your code.

3.1.1 Block Comments

Block comments generally apply to some (or all) code that follows them, and are indented to the same level as that code. Each line of a block comment starts with a `#` and a single space (unless it is indented text inside the comment).

Paragraphs inside a block comment are separated by a line containing a single #.

3.1.2 Inline Comments

Use inline comments sparingly.

An inline comment is a comment on the same line as a statement. Inline comments should be separated by at least two spaces from the statement. They should start with a `#` and a single space.

Inline comments are unnecessary and in fact distracting if they state the obvious. Don't do this:

```
x = x + 1                 # Increment x
```

But sometimes, this is useful:

```
x = x + 1                 # Compensate for border
```

3.1.3 Docstrings

- Write docstrings for all public modules, functions, classes, and methods.
- Docstrings are not necessary for non-public methods, but you should have a comment that describes what the method does. This comment should appear after the def line.
- Surround docstrings with three double quotes on either side, as in """This is a docstring""".
- Put the """ that ends a multiline docstring on a line by itself:

```
"""Return a foobang

Optional plotz says to frobnicate the bizbaz first.
"""
```

- For one liner docstrings, please keep the closing """ on the same line:

```
"""Return an ex-parrot."""
```

- Recommend to use Numpy Docstring format. An example from numpy:

```
"""Docstring for the example.py module.

Modules names should have short, all-lowercase names.  The module name may
have underscores if this improves readability.

Every module should have a docstring at the very top of the file.  The
module's docstring may extend over multiple lines.  If your docstring does
extend over multiple lines, the closing three quotation marks must be on
a line by itself, preferably preceded by a blank line.

"""
from __future__ import division, absolute_import, print_function

import os  # standard library imports first

# Do NOT import using *, e.g. from numpy import *
#
# Import the module using
#
#   import numpy
#
# instead or import individual functions as needed, e.g
#
```

```
#  from numpy import array, zeros
#
# If you prefer the use of abbreviated module names, we suggest the
# convention used by NumPy itself::

import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt

# These abbreviated names are not to be used in docstrings; users must
# be able to paste and execute docstrings after importing only the
# numpy module itself, unabbreviated.


def foo(var1, var2, *args, long_var_name='hi', only_seldom_used_keyword=0, **kwargs):
    r"""Summarize the function in one line.

    Several sentences providing an extended description. Refer to
    variables using back-ticks, e.g. `var`.

    Parameters
    ----------
    var1 : array_like
        Array_like means all those objects -- lists, nested lists, etc. --
        that can be converted to an array.  We can also refer to
        variables like `var1`.
    var2 : int
        The type above can either refer to an actual Python type
        (e.g. ``int``), or describe the type of the variable in more
        detail, e.g. ``(N,) ndarray`` or ``array_like``.
    *args : iterable
        Other arguments.
    long_var_name : {'hi', 'ho'}, optional
        Choices in brackets, default first when optional.

    Returns
    -------
    type
        Explanation of anonymous return value of type ``type``.
    describe : type
        Explanation of return value named `describe`.
    out : type
        Explanation of `out`.
    type_without_description
```

```
    Other Parameters
    ----------------
    only_seldom_used_keyword : int, optional
        Infrequently used parameters can be described under this optional
        section to prevent cluttering the Parameters section.
    **kwargs : dict
        Other infrequently used keyword arguments. Note that all keyword
        arguments appearing after the first parameter specified under the
        Other Parameters section, should also be described under this
        section.

    Raises
    ------
    BadException
        Because you shouldn't have done that.

    See Also
    --------
    numpy.array : Relationship (optional).
    numpy.ndarray : Relationship (optional), which could be fairly long, in
                    which case the line wraps here.
    numpy.dot, numpy.linalg.norm, numpy.eye

    Notes
    -----
    Notes about the implementation algorithm (if needed).

    This can have multiple paragraphs.

    You may include some math:

    .. math:: X(e^{j\omega } ) = x(n)e^{ - j\omega n}

    And even use a Greek symbol like :math:`\omega` inline.

    References
    ----------
    Cite the relevant literature, e.g. [1]_.  You may also cite these
    references in the notes section above.

    .. [1] O. McNoleg, "The integration of GIS, remote sensing,
       expert systems and adaptive co-kriging for environmental habitat
       modelling of the Highland Haggis using object-oriented, fuzzy-logic
       and neural-network techniques," Computers & Geosciences, vol. 22,
       pp. 585-588, 1996.
```

```
    Examples
    --------
    These are written in doctest format, and should illustrate how to
    use the function.

    >>> a = [1, 2, 3]
    >>> print([x + 3 for x in a])
    [4, 5, 6]
    >>> print("a\nb")
    a
    b
    """
    # After closing class docstring, there should be one blank line to
    # separate following codes (according to PEP257).
    # But for function, method and module, there should be no blank lines
    # after closing the docstring.
    pass
```

3.2 Naming Conventions

`module_name`, `package_name`, `ClassName`, `method_name`, `ExceptionName`, `function_name`, `GLOBAL_CONSTANT_NAME`, `global_var_name`, `instance_var_name`, `function_parameter_name`, `local_var_name`.

Function names, variable names, and filenames should be descriptive. In particular, do not use abbreviations that are ambiguous or unfamiliar to readers outside your project, and do not abbreviate by deleting letters within a word.

3.2.1 Names to Avoid

- single character names, except for specifically allowed cases:
  - counters or iterators (e.g. `i`, `j`, `k`, `v`, et al.).
  - `e` as an exception identifier in try/except statements.
  - `f` as a file handle in with statements.
- dashes (`-`) in any package/module name.
- `__double_leading_and_trailing_underscore__` names (reserved by Python).

3.2.2 File Naming

Python filenames must have a `.py` extension and must not contain dashes (`-`).

3.2.2.1 Naming guides from google python style

Type	Public	Internal
Packages	`lower_with_under`	
Classes	`CapWords`	`_CapWords`
Exceptions	`CapWords`	
Functions	`lower_with_under()`	`lower_with_under()`
Global/Class Constants	`CAPS_WITH_UNDER`	`_CAPS_WITH_UNDER`
Global/Class Variables	`lower_with_under`	`_lower_with_under`
Instance Variables	`lower_with_under`	`_lower_with_under`
Method Names	`lower_with_under()`	`_lower_with_under()`
Function/Method Parameters	`lower_with_under`	
Local Variables	`lower_with_under`	

3.3 Code Layout

3.3.1 Line length

- Maximum line length is 80 characters.
- Explicit exceptions to the 80 character limit:
  - Long import statements.
  - URLs, pathnames, or long flags in comments.

```
# long URL on their own line if necessary.
# http:/www.example.com/longurl/xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

- When a literal string won't fit on a single line, use parentheses for implicit line joining.

```
x = ('This will build a very long long '
     'long long long long long long string')
```

- Python will assume line continuation if code is contained within parentheses, brackets, or braces:

```
def function(arg_one, arg_two,
             arg_three, arg_four):
    return arg_one
```

- If it is impossible to use implied continuation, then you can use backslashes to break lines instead. However, if you can use implied continuation, then you should do so.

```
from mypkg import example1, \
    example2, example3
```

- If line breaking needs to occur around binary operators, like + and *, it should occur before the operator.

```
total = (first_variable
         + second_variable
         - third_variable)
```

3.3.2 Indentation

- Use 4 spaces per indentation level.
- Prefer spaces over tabs.
- Never use tabs or mix tabs and spaces.

```
# Aligned with opening delimiter.
foo = long_function_name(var_one, var_two,
                         var_three, var_four)

# Add 4 spaces (an extra level of indentation) to distinguish arguments from
# the rest.
def long_function_name(
        var_one, var_two, var_three,
        var_four):
    print(var_one)

# Hanging indents should add a level.
foo = long_function_name(
    var_one, var_two,
    var_three, var_four)
```

3.3.3 Parentheses

- Use parentheses around tuples.
- Do not use them in return statements or conditional statements unless using parentheses for implied line continuation or to indicate a tuple.

```
# good
if foo:
    bar()
while x:
    x = bar()
if x and y:
    bar()
if not x:
    bar()
# For a 1 item tuple the ()s are more visually obvious than the comma.
onesie = (foo,)
return foo
```

```
return spam, beans
return (spam, beans)
for (x, y) in dict.items(): ...

# bad
if (x):
    bar()
if not(x):
    bar()
return (foo)
```

3.3.4 Blank Lines

- Two blank lines between top-level definitions, be they function or class definitions.

```
class MyFirstClass:
    pass


class MySecondClass:
    pass


def top_level_function():
    return None
```

- One blank line between method definitions inside classes.

```
class MyClass:
    def first_method(self):
        return None

    def second_method(self):
        return None
```

- Use single blank lines as you judge appropriate within functions or methods.

3.3.5 Whitespace

- No whitespace inside parentheses, brackets or braces.

```
# good
spam(ham[1], {'eggs': 2}, [])

# bad
spam( ham[ 1 ], { eggs: 2 } )
```

- No whitespace before a comma, semicolon, or colon. Do use whitespace after a comma, semicolon, or colon, except at the end of the line.

```
# good
if x == 4:
    print x, y
x, y = y, x

# bad
if x == 4 :
    print x , y
x , y = y , x
```

- No whitespace before the open paren/bracket that starts an argument list, indexing or slicing.

```
# good
spam(1)
dict['key'] = list[index]

# bad
spam (1)
dict ['key'] = list [index]
```

- Always surround these binary operators with a single space on either side: assignment (=), augmented assignment (+=, -= etc.), comparisons (==, <, >, !=, <>, <=, >=, in, not in, is, is not), Booleans (and, or, not).

```
# good
i = i + 1
submitted += 1
x = x*2 - 1
hypot2 = x*x + y*y
c = (a+b) * (a-b)

# bad
i=i+1
submitted +=1
x = x * 2 - 1
hypot2 = x * x + y * y
c = (a + b) * (a - b)
```

- Never use spaces around = when passing keyword arguments or defining a default parameter value, with one exception: when a type annotation is present, do use spaces around the = for the default parameter value.

```
# good
def complex(real, imag=0.0): return Magic(r=real, i=imag)
def complex(real, imag: float = 0.0): return Magic(r=real, i=imag)
```

```
# bad
def complex(real, imag = 0.0): return Magic(r = real, i = imag)
def complex(real, imag: float=0.0): return Magic(r = real, i = imag)
```

3.3.6 Trailing Commas

Trailing commas in sequences of items are recommended only when the closing container token `]`, `)`, or `}` does not appear on the same line as the final element.

```
# good
golomb3 = [0, 1, 3]
golomb4 = [
    0,
    1,
    4,
    6,
]

# bad
golomb4 = [0, 1, 4, 6,]
```

3.3.7 Strings

- Use an `f-string`, the `%` operator, or the `format` method for formatting strings, even when the parameters are all strings.
- Use your best judgment to decide between + and % (or format) though.
- Do not use % or the format method for pure concatenation.

```
# good
x = '%s, %s!' % (imperative, expletive)
x = '{}, {}'.format(first, second)
x = 'name: %s; score: %d' % (name, n)
x = 'name: {}; score: {}'.format(name, n)
x = f'name: {name}; score: {n}'

# bad
x = '%s%s' % (a, b)  # use + in this case
x = '{}{}'.format(a, b)  # use + in this case
x = first + ', ' + second
x = 'name: ' + name + '; score: ' + str(n)
```

- Be consistent with your choice of string quote character within a file. Pick `'` or `"` and stick with it. It is okay to use the other quote character on a string to avoid the need to backslash-escape quote characters within the string.

- Prefer """ for multi-line strings rather than '''.
- Multi-line strings do not flow with the indentation of the rest of the program. If you need to avoid embedding extra space in the string, use concatenated single-line strings.

```
# good
long_string = """This is fine if your use case can accept
    extraneous leading spaces."""
long_string = ("And this is fine if you cannot accept\n" +
               "extraneous leading spaces.")

# bad
long_string = """This is pretty ugly.
Don't do this.
"""
```

3.3.8 Imports

- Imports should usually be on separate lines.

```
# good
import os
import sys

# bad
import sys, os
```

- Imports are always put at the top of the file, just after any module comments and docstrings and before module globals and constants. Imports should be grouped in the following order, and you should put a blank line between each group of imports.
  - Standard library imports.
  - Related third party imports.
  - Local application/library specific imports.

```
import collections
import sys

from absl import app
from absl import flags

from myproject.backend import huxley
from myproject.backend.state_machine import main_loop
```

3.4 References

- PEP 8 style guide for python code
- The elements of python style
- Google python style guid
- numpy Docstring format

Chapter 4

R Style Guide

4.1 Files

4.1.1 Names

- File names should be meaningful and end in .R. Avoid using special characters in file names - stick with numbers, letters, -, and _.

```
# Good
fit_models.R
utility_functions.R

# Bad
fit models.R
foo.r
stuff.r
```

- If files should be run in a particular order, prefix them with numbers. If it seems likely you'll have more than 10 files, left pad with zero:

```
00_download.R
01_explore.R
...
09_model.R
10_visualize.R
```

- Prefer file names that are all lower case, and never have names that differ only in their capitalization.

4.1.2 Internal structure

- Use commented lines of - and = to break up your file into easily readable chunks.

```
# Load data ---------------------------

# Plot data ---------------------------
```

- If your script uses add-on packages, load them all at once at the very beginning of the file.

4.2 Naming Convention

- Variable and function names should use only lowercase letters, numbers, and _. Use underscores (_) (so called snake case) to separate words within a name.

```
# Good
day_one
day_1
```

- It's better to reserve dots exclusively for the S3 object system. In S3, methods are given the name `function.class`.

- Generally, variable names should be nouns and function names should be verbs. Strive for names that are concise and meaningful (this is not easy!).

```
# Good
day_one

# Bad
first_day_of_the_month
djm1
```

- Avoid re-using names of common functions and variables. This will cause confusion for the readers of your code.

```
# Bad
T <- FALSE
c <- 10
mean <- function(x) sum(x)
```

4.3 Spacing

4.3.1 Indentation

Use four spaces.

4.3.2 Commas

Always put a space after a comma, never before, just like in regular English.

```
# Good
x[, 1]

# Bad
x[,1]
x[ ,1]
x[ , 1]
```

4.3.3 Parentheses

- Do not put spaces inside or outside parentheses for regular function calls.

```
# Good
mean(x, na.rm = TRUE)

# Bad
mean (x, na.rm = TRUE)
mean( x, na.rm = TRUE )
```

- Place a space before and after `()` when used with `if`, `for`, or `while`.

```
# Good
if (debug) {
    show(x)
}

# Bad
if(debug){
    show(x)
}
```

- Place a space after `()` used for function arguments:

```
# Good
function(x) {}

# Bad
function (x) {}
function(x){}
```

4.3.4 Embracing

The embracing operator, `{{ }}`, should always have inner spaces to help emphasise its special behaviour:

```
# Good
max_by <- function(data, var, by) {
  data %>%
    group_by({{ by }}) %>%
    summarise(maximum = max({{ var }}, na.rm = TRUE))
}

# Bad
max_by <- function(data, var, by) {
  data %>%
    group_by({{by}}) %>%
    summarise(maximum = max({{var}}, na.rm = TRUE))
}
```

4.3.5 Infix operators

Most infix operators (`==`, `+`, `-`, `<-`, etc.) should always be surrounded by spaces:

```
# Good
height <- (feet * 12) + inches
mean(x, na.rm = TRUE)

# Bad
height<-feet*12+inches
mean(x, na.rm=TRUE)
```

There are a few exceptions, which should never be surrounded by spaces:

- The operators with high precedence: `::`, `:::`, `$`, `@`, `[`, `[[`, `^`, unary `-`, unary `+`, and `:`.
- Single-sided formulas when the right-hand side is a single identifier.

  ```
  # Good
  ~foo

  # Bad
  ~ foo
  ```

  Note that single-sided formulas with a complex right-hand side do need a space:

  ```
  # Good
  ~ .x + .y

  # Bad
  ~.x + .y
  ```

- When used in tidy evaluation `!!` (bang-bang) and `!!!` (bang-bang-bang)

4.4 Function Calls

4.4.1 Named arguments

- If you override the default value of an argument, use the full name.
- You can omit the argument names for very common arguments

```
# Good
mean(1:10, na.rm = TRUE)

# Bad
mean(x = 1:10, , FALSE)
mean(, TRUE, x = c(1:10, NA))
```

4.4.2 Assignment

Avoid assignment in function calls:

```
# Good
x <- complicated_function()
if (nzchar(x) < 1) {
  # do something
}

# Bad
if (nzchar(x <- complicated_function()) < 1) {
  # do something
}
```

The only exception is in functions that capture side-effects:

```
output <- capture.output(x <- f())
```

4.4.3 Long lines

There are two options if the function name and definition can't fit on a single line:

- Function-indent: place each argument on its own line, and indent to match the opening `(` of function:

```
long_function_name <- function(a = "a long argument",
                               b = "another argument",
                               c = "another long argument") {
  # As usual code is indented by two spaces.
}
```

- Double-indent: Place each argument of its own double indented line.

```
long_function_name <- function(
    a = "a long argument",
    b = "another argument",
    c = "another long argument") {
    # As usual code is indented by two spaces.
}
```

In both cases the trailing `)` and leading `{` should go on the same line as the last argument.

Prefer function-indent style to double-indent style when it fits.

These styles are designed to clearly separate the function definition from its body.

```
# Bad
long_function_name <- function(a = "a long argument",
    b = "another argument",
    c = "another long argument") {
    # Here it's hard to spot where the definition ends and the
    # code begins, and to see all three function arguments
}
```

4.4.4 Return

- Only use `return()` for early returns. Otherwise, rely on R to return the result of the last evaluated expression.

```
# Good
find_abs <- function(x) {
    if (x > 0) {
        return(x)
    }
    x * -1
}
add_two <- function(x, y) {
    x + y
}

# Bad
add_two <- function(x, y) {
    return(x + y)
}
```

- Return statements should always be on their own line because they have important effects on the control flow

```r
# Good
find_abs <- function(x) {
    if (x > 0) {
        return(x)
    }
    x * -1
}

# Bad
find_abs <- function(x) {
    if (x > 0) return(x)
    x * -1
}
```

- If your function is called primarily for its side-effects (like printing, plotting, or saving to disk), it should return the first argument invisibly. This makes it possible to use the function as part of a pipe. print methods should usually do this, like this example from httr:

```r
print.url <- function(x, ...) {
    cat("Url: ", build_url(x), "\n", sep = "")
    invisible(x)
}
```

4.4.5 Comments

- In code, use comments to explain the "why" not the "what" or "how". Each line of a comment should begin with the comment symbol and a single space: #.

```r
# Good

# Objects like data frames are treated as leaves
x <- map_if(x, is_bare_list, recurse)


# Bad

# Recurse only with bare lists
x <- map_if(x, is_bare_list, recurse)
```

- Comments should be in sentence case, and only end with a full stop if they contain at least two sentences:

```r
# Good

# Objects like data frames are treated as leaves
```

```
x <- map_if(x, is_bare_list, recurse)

# Do not use `is.list()`. Objects like data frames must be treated
# as leaves.
x <- map_if(x, is_bare_list, recurse)


# Bad

# objects like data frames are treated as leaves
x <- map_if(x, is_bare_list, recurse)

# Objects like data frames are treated as leaves.
x <- map_if(x, is_bare_list, recurse)
```

4.5 Control flow

4.5.1 Code blocks

- `{` should be the last character on the line. Related code (e.g., an `if` clause, a function declaration, a trailing comma, …) must be on the same line as the opening brace.
- The contents should be indented by four spaces.
- `}` should be the first character on the line.

```
# good
if (y < 0 && debug) {
    message("y is negative")
}

if (y == 0) {
    if (x > 0) {
        log(x)
    } else {
        message("x is negative or zero")
    }
} else {
    y^x
}

# bad
if (y == 0)
{
  if (x > 0) {
```

```
    log(x)
  } else {
message("x is negative or zero")
  }
} else { y ^ x }
```

4.5.2 If Statements

- If used, `else` should be on the same line as `}`.
- `&` and `|` should never be used inside of an `if` clause because they can return vectors. Always use `&&` and `||` instead.

4.5.3 Inline statement

- You can write a simple `if` block on ane single line

```
message <- if (x > 10) "big" else "small"
```

- Function calls that affect control flow (like `return()`, `stop()` or `continue`) should always go in their own `{}` block:

```
# Good
if (y < 0) {
    stop("Y is negative")
}

find_abs <- function(x) {
    if (x > 0) {
        return(x)
    }
    x * -1
}

# Bad
if (y < 0) stop("Y is negative")

if (y < 0)
    stop("Y is negative")
```

4.5.4 Implicit type coercion

Avoid implicit type coercion (e.g. from numeric to logical) in `if` statements:

```
# Good
if (length(x) > 0) {
    # do something
}
```

```
# Bad
if (length(x)) {
    # do something
}
```

4.5.5 Switch statements

- Avoid position-based `switch()` statements (i.e. prefer names).
- Each element should go on its own line.
- Elements that fall through to the following element should have a space after `=`.
- Provide a fall-through error, unless you have previously validated the input.

```
# Good
switch(x,
    a = ,
    b = 1,
    c = 2,
    stop("Unknown `x`", call. = FALSE)
)

# Bad
switch(x, a = , b = 1, c = 2)
switch(x, a =, b = 1, c = 2)
switch(y, 1, 2, 3)
```

4.6 Long lines

- Strive to limit your code to 80 characters per line.
- If a function call is too long to fit on a single line, use one line each for the function name, each argument, and the closing `)`. This makes the code easier to read and to change later.

```
# Good
do_something_very_complicated(
    something = "that",
    requires = many,
    arguments = "some of which may be long"
)

# Bad
do_something_very_complicated("that", requires, many, arguments,
```

```
                                  "some of which may be long"
                                )
```

- Short unnamed arguments can also go on the same line as the function name, even if the whole function call spans multiple lines.

```
map(x, f,
    extra_argument_a = 10,
    extra_argument_b = c(1, 43, 390, 210209)
)
```

- You may also place several arguments on the same line if they are closely related to each other.

```
# Good
paste0(
    "Requirement: ", requires, "\n",
    "Result: ", result, "\n"
)

# Bad
paste0(
    "Requirement: ", requires,
    "\n", "Result: ",
    result, "\n")
```

4.7 Semicolons

Don't put `;` at the end of a line, and don't use `;` to put multiple commands on one line.

4.8 Assignment

Use `<-`, not `=`, for assignment.

4.9 Character vectors

Use `"`, not `'`, for quoting text. The only exception is when the text already contains double quotes and no single quotes.

```
# Good
"Text"
'Text with "quotes"'
'<a href="http://style.tidyverse.org">A link</a>'
```

```
# Bad
'Text'
'Text with "double" and \'single\' quotes'
```

4.10 Logical vectors

Use `TRUE` and `FALSE` rather than `T` and `F`.

4.11 Comments

Each line of a comment should begin with the comment symbol `#` and a single space.

In data analysis code, use comments to record important findings and analysis decisions. If you need comments to explain what your code is doing, consider rewriting your code to be clearer. If you discover that you have more comments than code, consider switching to R Markdown.

4.12 Resources

- styler: formatting your code according to the tidyverse style guide (or your custom style guide) so you can direct your attention to the content of your code. It helps to keep the coding style consistent across projects and facilitate collaboration.
- lintr: offering static code analysis for R. It checks adherence to a given style, syntax errors and possible semantic issues.