Good coding practices

in  and  + 

Caterina Constantinescu, EdinbR organiser

caterina.constantinescu@ed.ac.uk

November 18, 2015

Outline

1. **General principles**

2. **Major topics regardless of language**
   - General form
   - Comments
   - Imports
   - White space
   - Naming conventions
   - Writing functions

Outline

Regardless of the language:

- Code is read much more often than it is written (PEP8).

Regardless of the language:

- Code is read much more often than it is written (PEP8).
- Consistency, in order of importance:

Regardless of the language:

- Code is read much more often than it is written (PEP8).
- Consistency, in order of importance:
  1. within one module/function(/R script).

Regardless of the language:

- Code is read much more often than it is written (PEP8).
- Consistency, in order of importance:
  1. within one module/function(/R script).
  2. within a project.

Regardless of the language:

- Code is read much more often than it is written (PEP8).
- Consistency, in order of importance:
  1. within one module/function(/R script).
  2. within a project.
  3. with the generally accepted standard for that language.*

Outline

General form 1: File naming

- *.R

General form 1: File naming

- *.R
- Meaningful.

General form 1: File naming

- *.R
- Meaningful.
- Sensible directory, not Desktop.

General form 1: File naming

- *.R
- Meaningful.
- Sensible directory, not Desktop.
- Never use file names that differ only in capitalisation.

General form 1: File naming

- *.R
- Meaningful.
- Sensible directory, not Desktop.
- Never use file names that differ only in capitalisation.
- Prefixing script names with numbers can be useful for running in sequence.

General form 2: File organisation

1. Header comments: author, copyright notice, purpose (& maybe date, revisions) etc.

General form 2: File organisation

1. Header comments: author, copyright notice, purpose (& maybe date, revisions) etc.
2. Package declarations (imports) or source(script): first non-comment line in a script.

General form 2: File organisation

1. Header comments: author, copyright notice, purpose (& maybe date, revisions) etc.
2. Package declarations (imports) or source(script): first non-comment line in a script.
3. Custom functions/methods.

General form 2: File organisation

1. Header comments: author, copyright notice, purpose (& maybe date, revisions) etc.
2. Package declarations (imports) or source(script): first non-comment line in a script.
3. Custom functions/methods.
4. Constants.

General form 2: File organisation

1. Header comments: author, copyright notice, purpose (& maybe date, revisions) etc.
2. Package declarations (imports) or source(script): first non-comment line in a script.
3. Custom functions/methods.
4. Constants.
5. Object declarations (and read-in data sets).

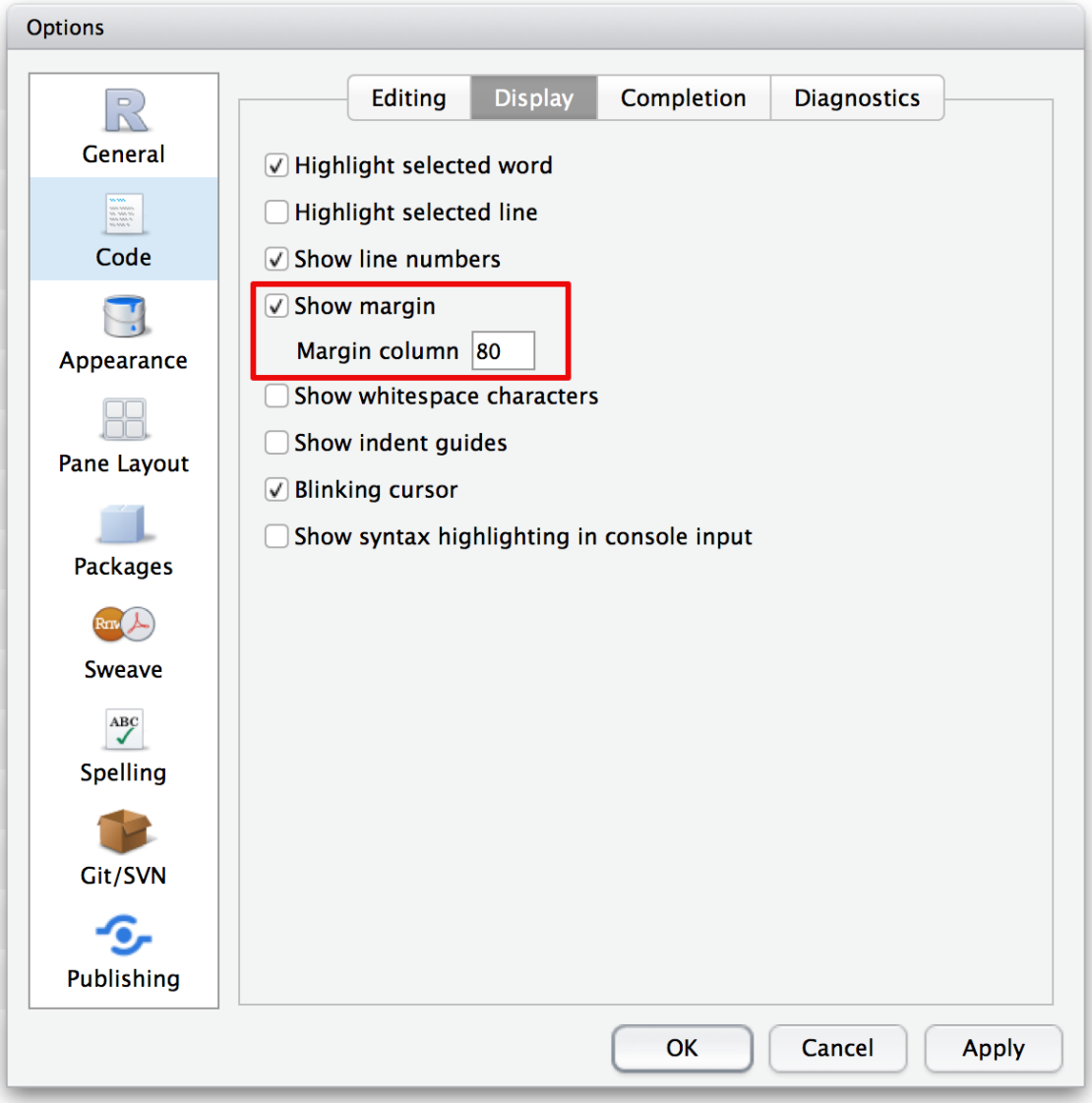
General form 2: File organisation

1. Header comments: author, copyright notice, purpose (& maybe date, revisions) etc.
2. Package declarations (imports) or source(script): first non-comment line in a script.
3. Custom functions/methods.
4. Constants.
5. Object declarations (and read-in data sets).
6. Executed statements (analysis).

General form 2: File organisation

```r
# Author: Caterina Constantinescu
# 12 November 2015
# EdinbR presentation due on 18 November 2015

# Imports ----------------------------------------------------------------------

library(datasets)

# Functions --------------------------------------------------------------------

my_function <- function(dataset){
  return(names(dataset))
}

# Constants --------------------------------------------------------------------

IMPORTANT_COLUMNS <- c("hp", "cyl")

# Datasets / Object declarations -----------------------------------------------

data(mtcars)
my_data <- mtcars

# Analysis ---------------------------------------------------------------------

my_function(my_data)
```

General form 3: Line length



Comments

- Explain the why, not the what.

Comments

- Explain the why, not the what.
- Comments that contradict the code = worse than no comments (PEP8). So, keep them up to date.

Comments

- Explain the why, not the what.
- Comments that contradict the code = worse than no comments (PEP8). So, keep them up to date.
- They should normally be complete sentences.

Comments

- Explain the why, not the what.
- Comments that contradict the code = worse than no comments (PEP8). So, keep them up to date.
- They should normally be complete sentences.
- English everywhere.

Comments

- Explain the why, not the what.
- Comments that contradict the code = worse than no comments (PEP8). So, keep them up to date.
- They should normally be complete sentences.
- English everywhere.
- Inline/trailing comments - use sparingly.

Comments

- Explain the why, not the what.
- Comments that contradict the code = worse than no comments (PEP8). So, keep them up to date.
- They should normally be complete sentences.
- English everywhere.
- Inline/trailing comments - use sparingly.
- Indent at the same level as surrounding code.

Comments

- Explain the why, not the what.
- Comments that contradict the code = worse than no comments (PEP8). So, keep them up to date.
- They should normally be complete sentences.
- English everywhere.
- Inline/trailing comments - use sparingly.
- Indent at the same level as surrounding code.
- Temporarily removing code: ONLY for active development phase.

Imports

- Usually on separate lines.

Imports

- Usually on separate lines.
- Order: standard, related ones, very specific ones + blank line between these categories.

Imports

- Usually on separate lines.
- Order: standard, related ones, very specific ones + blank line between these categories.
- Do not import things you don't use.

Imports

- Usually on separate lines.
- Order: standard, related ones, very specific ones + blank line between these categories.
- Do not import things you don't use.

```
1 library(utils) # getAnywhere() lives here
2
3 library(descr)
4 library(psych)
5 library(Hmisc)
6
7 library(lme4)
```

White space 1a. Blank spaces

```
1  # Comment
2
3  # Always after a comma, but not between function and bracket,
4  # or bracket and first arg:
5  c(1, 2, 3)
6
7  # No space between '=' sign and args in fun call:
8  some_function(arg1=FALSE, arg2=TRUE)
9
10 # Between keyword and parenthesis:
11 if (some_number == 2)
12
13 base::get
14
15 some_variable[1, 5] # But:
16 some_other_variable[1:5]
17
18 my_number <- 1
19
20 my_average <- mean(my_vector1 / my_vector2 + 3)
```

White space 1b. Blank lines

- Can be used to delimit logical sections / related function groups.

White space 1b. Blank lines

- Can be used to delimit logical sections / related function groups.
- Can be omitted between related one-liners.

White space 2. Indentation

- Spaces recommended, rather than tabs.

White space 2. Indentation

- Spaces recommended, rather than tabs.
- Typically, 1 unit of indentation = 4 spaces (or 2 spaces).

White space 2. Indentation

- Spaces recommended, rather than tabs.
- Typically, 1 unit of indentation = 4 spaces (or 2 spaces).
- Don't use spaces manually, rely on the editor's automatic wrapping of lines within () or {}.

White space 2. Indentation

- Spaces recommended, rather than tabs.
- Typically, 1 unit of indentation = 4 spaces (or 2 spaces).
- Don't use spaces manually, rely on the editor's automatic wrapping of lines within () or {}.
- Don't mix tabs and spaces when indenting.

White space 2. Indentation

- Spaces recommended, rather than tabs.
- Typically, 1 unit of indentation = 4 spaces (or 2 spaces).
- Don't use spaces manually, rely on the editor's automatic wrapping of lines within () or {}.
- Don't mix tabs and spaces when indenting.
- Wrap lines after e.g., comma.

Naming styles

1. Styles
   - lowercase

Naming styles

1. Styles
   - lowercase
   - lowercase_with_underscores

Naming styles

1. Styles
   - lowercase
   - lowercase_with_underscores
   - UPPERCASE

Naming styles

1. Styles
   - lowercase
   - lowercase_with_underscores
   - UPPERCASE
   - UPPERCASE_WITH_UNDERSCORES

Naming styles

1. Styles
   - lowercase
   - lowercase_with_underscores
   - UPPERCASE
   - UPPERCASE_WITH_UNDERSCORES
   - mixedCase

Naming styles

1. Styles
   - lowercase
   - lowercase_with_underscores
   - UPPERCASE
   - UPPERCASE_WITH_UNDERSCORES
   - mixedCase
   - CapitalisedWords or CamelCase

Naming styles

1. Styles
   - lowercase
   - lowercase_with_underscores
   - UPPERCASE
   - UPPERCASE_WITH_UNDERSCORES
   - mixedCase
   - CapitalisedWords or CamelCase

2. Misc
   - Never use 'l'(el), 'O' (oh), or 'I' ('eye') as variable names - in some fonts they are not easily distinguishable from 0 / 1.

Naming styles

1. Styles
   - lowercase
   - lowercase_with_underscores
   - UPPERCASE
   - UPPERCASE_WITH_UNDERSCORES
   - mixedCase
   - CapitalisedWords or CamelCase

2. Misc
   - Never use 'l'(el), 'O' (oh), or 'I' ('eye') as variable names - in some fonts they are not easily distinguishable from 0 / 1.
   - Object names should be meaningful, and when possible, concise.

Naming conventions

- Functions (verbs): Preferred: do_something_with_input() (Python and Hadley).

Naming conventions

- Functions (verbs): Preferred: do_something_with_input() (Python and Hadley).
  - Alternative: DoSomethingWithInput() or doSomethingWithInput() - preferred by Google or Bioconductor.

Naming conventions

- Functions (verbs): Preferred: do_something_with_input() (Python and Hadley).
  - Alternative: DoSomethingWithInput() or doSomethingWithInput() - preferred by Google or Bioconductor.
  - Java: get and set: someClass.getURLProtocol() or setPhoneNumber().

Naming conventions

- Functions (verbs): Preferred: do_something_with_input() (Python and Hadley).
  - Alternative: DoSomethingWithInput() or doSomethingWithInput() - preferred by Google or Bioconductor.
  - Java: get and set: someClass.getURLProtocol() or setPhoneNumber().
- CONSTANTS: TOTAL, MAX_OVERFLOW.

Naming conventions

- Functions (verbs): Preferred: do_something_with_input() (Python and Hadley).
  - Alternative: DoSomethingWithInput() or doSomethingWithInput() - preferred by Google or Bioconductor.
  - Java: get and set: someClass.getURLProtocol() or setPhoneNumber().
- CONSTANTS: TOTAL, MAX_OVERFLOW.
- Variables (nouns): some_variable (disliked by Google and Bioconductor), who suggest someVariable or some.variable*.

Writing functions

1. Curly braces
   - { never alone on line, e.g.: if (y < 0) { ...blank line

Writing functions

1. Curly braces
   - { never alone on line, e.g.: if (y < 0) { ...blank line
   - } normally should be alone on line*

Writing functions

1. Curly braces
   - { never alone on line, e.g.: if (y < 0) { ...blank line
   - } normally should be alone on line*
   - } else {

Writing functions

1. Curly braces
   - { never alone on line, e.g.: if (y < 0) { ...blank line
   - } normally should be alone on line*
   - } else {
2. return() before final brace

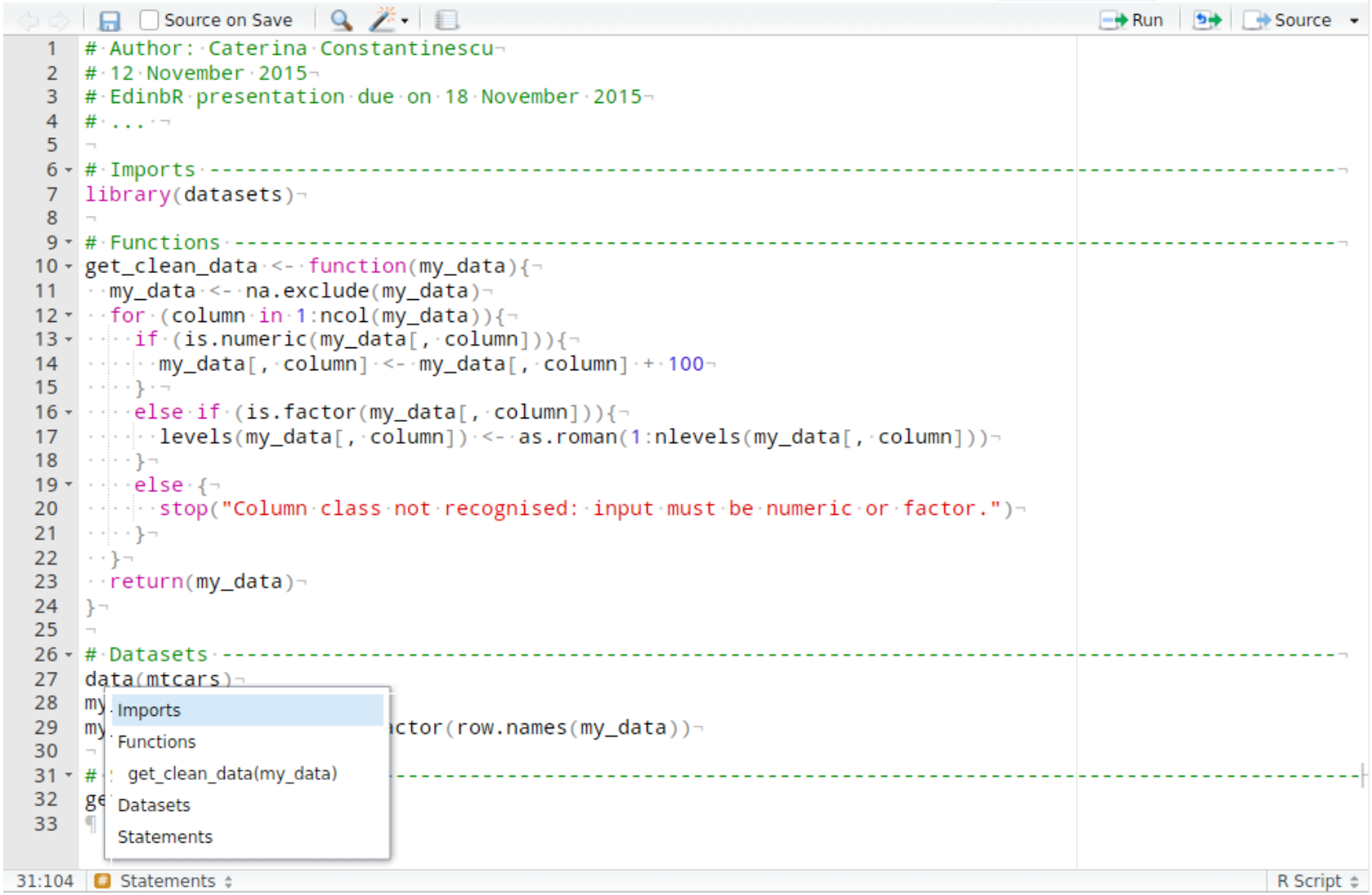
Writing functions

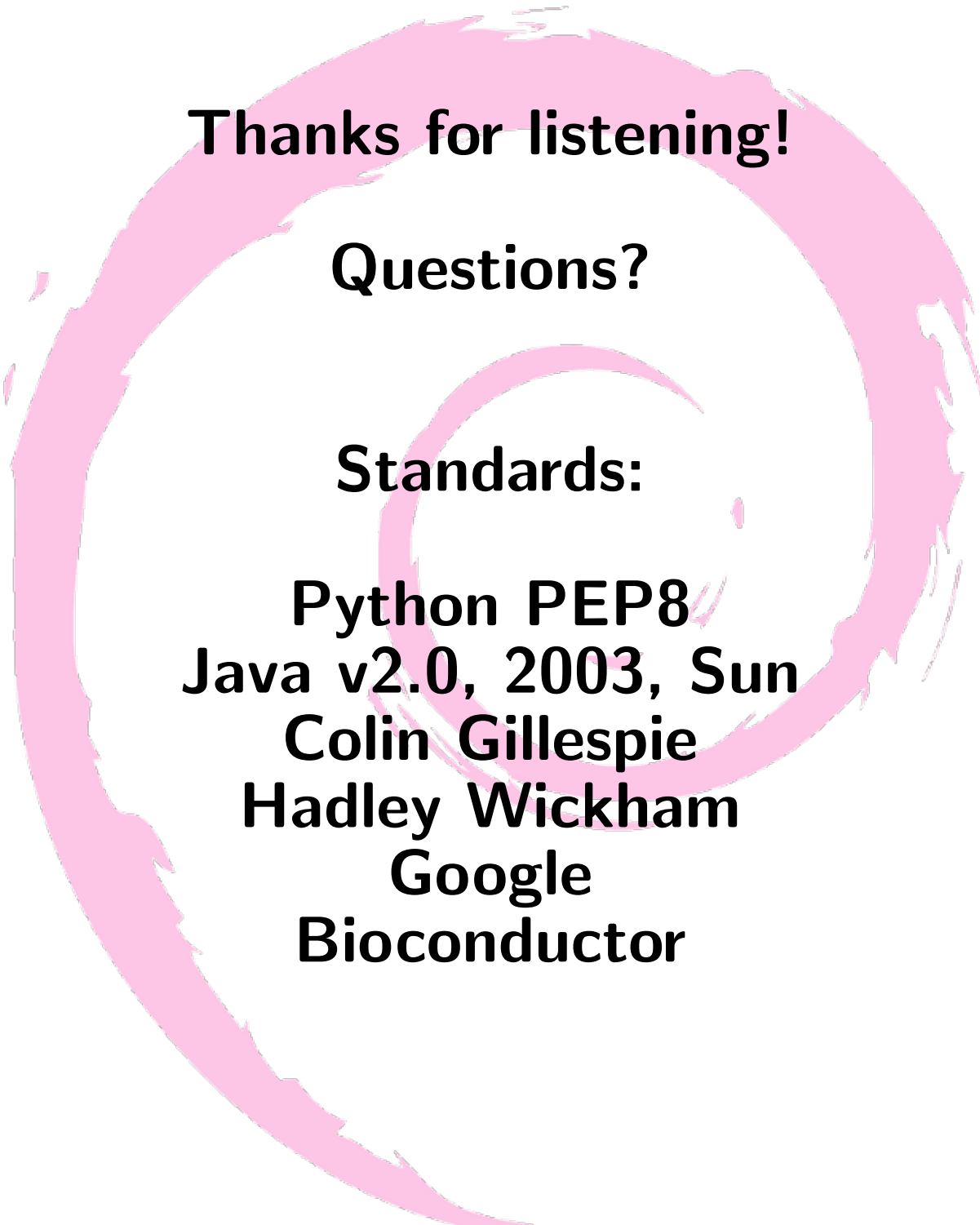
1. Curly braces
   - { never alone on line, e.g.: if (y < 0) { ...blank line
   - } normally should be alone on line*
   - } else {
2. return() before final brace
3. Length: 20-30 lines

Example

```r
# Author: Caterina Constantinescu
# EdinbR presentation due on 18 November 2015

# Imports ----------------------------------------------------------------------
library(datasets)

# Functions --------------------------------------------------------------------
get_clean_data <- function(my_data){
  my_data <- na.exclude(my_data)
  for (column in 1:ncol(my_data)){
    if (is.numeric(my_data[, column])){
      my_data[, column] <- my_data[, column] + 100
    }
    else if (is.factor(my_data[, column])){
      levels(my_data[, column]) <- as.roman(1:nlevels(my_data[, column]))
    }
    else {
      stop("Column class not recognised: input must be numeric or factor.")
    }
  }
  return(my_data)
}

# Datasets ---------------------------------------------------------------------
data(mtcars)
my_data <- mtcars
my_data$car_type <- as.factor(row.names(my_data))

# Statements -------------------------------------------------------------------
get_clean_data(my_data)
```

Example


Source on Save
Run
Source
# Author: Caterina Constantinescu
# 12 November 2015
# EdinbR presentation due on 18 November 2015
# ...
# Imports
library(datasets)
# Functions
get_clean_data <- function(my_data){
my_data <- na.exclude(my_data)
for (column in 1:ncol(my_data)){
if (is.numeric(my_data[, column])){
my_data[, column] <- my_data[, column] + 100
}
else if (is.factor(my_data[, column])){
levels(my_data[, column]) <- as.roman(1:nlevels(my_data[, column]))
}
else {
stop("Column class not recognised: input must be numeric or factor.")
}
}
return(my_data)
}
# Datasets
data(mtcars)
Imports
Functions
get_clean_data(my_data)
Datasets
Statements
31:104
Statements
R Script



Thanks for listening!

Questions?

Standards:

Python PEP8
Java v2.0, 2003, Sun
Colin Gillespie
Hadley Wickham
Google
Bioconductor