Building an R Package with LLMs

Jared P. Lander
Lander Analytics





What are we Doing?





ChatGPT

You are an experienced R programmer. Write a function whose input is a path to a folder and the output is a count of each of the different file types in the folder.

```r
fileTypeCounts <- function(dirPath) {
  # Ensure the directory exists
  if(!dir.exists(dirPath)) {
    stop("Directory not found")
  }

  # List all files in the directory
  fileList <- list.files(dirPath, full.names = TRUE)

  # Get file extensions
  fileExt <- tools::file_ext(fileList)

  # Create a table (frequency count) of file extensions
  fileTypeCounts <- table(fileExt)

  return(fileTypeCounts)
}
```





```
1 fileTypeCounts('example_files')
```

```
## fileExt
##  csv json  txt
##    3    1    2
```

ChatGPT

Can you rebuild it with an argument that let’s us specify which filetypes to consider?

```r
fileTypeCounts <- function(dirPath, fileTypes = NULL) {
  # Ensure the directory exists
  if(!dir.exists(dirPath)) {
    stop("Directory not found")
  }

  # List all files in the directory
  fileList <- list.files(dirPath, full.names = TRUE)

  # Get file extensions
  fileExt <- tools::file_ext(fileList)

  # If specific file types are provided, only keep those
  if (!is.null(fileTypes)) {
    fileExt <- fileExt[fileExt %in% fileTypes]
  }

  # Create a table (frequency count) of file extensions
  fileTypeCounts <- table(fileExt)

  return(fileTypeCounts)
}
```





```
1 fileTypeCounts('example_files')
```

```
## fileExt
##  csv json  txt
##    3    1    2
```

```
1 fileTypeCounts('example_files', fileTypes=c('csv', 'json'))
```

```
## fileExt
##  csv json
##    3    1
```





ChatGPT

Can you rewrite it to use the tidyverse?





```r
library(purrr)
library(dplyr)
library(fs)

fileTypeCounts <- function(dirPath, fileTypes = NULL) {
  # Ensure the directory exists
  if(!dir.exists(dirPath)) {
    stop("Directory not found")
  }
  # List all files in the directory
  fileList <- fs::dir_ls(dirPath, glob = "*")
  # Get file extensions
  fileExt <- purrr::map_chr(fileList, ~ fs::path_ext(.x))
  # If specific file types are provided, only keep those
  if (!is.null(fileTypes)) {
    fileExt <- fileExt[fileExt %in% fileTypes]
  }
  # Create a frequency count of file extensions
  fileTypeCounts <- fileExt %>%
    dplyr::as_tibble() %>%
    dplyr::group_by(value) %>%
    dplyr::tally(sort = TRUE) %>%
    dplyr::rename(File_Type = value, Count = n)
  return(fileTypeCounts)
}
```

```
fileTypeCounts('example_files', fileTypes=c('csv', 'json'))
```

```
## # A tibble: 2 × 2
##   File_Type Count
##   <chr>     <int>
## 1 csv           3
## 2 json          1
```

ChatGPT

If you are going to use package::function() then you don't need to load the packages at the beginning.

```r
fileTypeCounts <- function(dirPath, fileTypes = NULL) {
  # Ensure the directory exists
  if(!fs::dir_exists(dirPath)) {
    stop("Directory not found")
  }
  # List all files in the directory
  fileList <- fs::dir_ls(dirPath, glob = "*")

  # Get file extensions
  fileExt <- purrr::map_chr(fileList, ~ fs::path_ext(.x))

  # If specific file types are provided, only keep those
  if (!is.null(fileTypes)) {
    fileExt <- fileExt[fileExt %in% fileTypes]
  }

  # Create a frequency count of file extensions
  fileTypeCounts <- fileExt %>%
    tibble::as_tibble() %>%
    dplyr::group_by(value) %>%
    dplyr::tally(sort = TRUE) %>%
    dplyr::rename(File_Type = value, Count = n)

  return(fileTypeCounts)
}
```





You're absolutely correct. If we're specifying the package name each time we call a function from it (as in package::function()), we don't actually need to load the packages with library(). This can be especially useful if we want to avoid namespace conflicts between packages.

This version has the advantage of being more explicit about where each function comes from, which can be helpful in understanding the code and avoiding conflicts.





ChatGPT

I noticed you are listing all files then only keeping the filetypes we want, if we specified any. Can you change it so that when listing files we only search for the specified filetypes, if any?

```r
fileTypeCounts <- function(dirPath, fileTypes = NULL) {
  # Ensure the directory exists
  if(!fs::dir_exists(dirPath)) {
    stop("Directory not found")
  }
  # If specific file types are provided, modify the glob pattern accordingly
  if (!is.null(fileTypes)) {
    glob_pattern <- paste0("*.", fileTypes)
  } else {
    glob_pattern <- "*"
  }
  # List files in the directory according to the glob pattern
  fileList <- purrr::map(glob_pattern, ~ fs::dir_ls(dirPath, glob = .x)) %>% unlist()
  # Get file extensions
  fileExt <- purrr::map_chr(fileList, ~ fs::path_ext(.x))
  # Create a frequency count of file extensions
  fileTypeCounts <- fileExt %>%
    tibble::as_tibble() %>%
    dplyr::group_by(value) %>%
    dplyr::tally(sort = TRUE) %>%
    dplyr::rename(File_Type = value, Count = n)
  return(fileTypeCounts)
}
```



ChatGPT

Instead of using purrr::map() to apply fs::dir_ls() to each element in glob_pattern, instead make a proper regex for the filetypes to provide to glob_pattern.

```r
fileTypeCounts <- function(dirPath, fileTypes = NULL) {
  # Ensure the directory exists
  if(!fs::dir_exists(dirPath)) {
    stop("Directory not found")
  }

  # If specific file types are provided, modify the glob pattern accordingly
  if (!is.null(fileTypes)) {
    glob_pattern <- paste0("*.{", paste(fileTypes, collapse = ","), "}")
  } else {
    glob_pattern <- "*"
  }

  # List files in the directory according to the glob pattern
  fileList <- fs::dir_ls(dirPath, glob = glob_pattern)

  # Get file extensions
  fileExt <- purrr::map_chr(fileList, ~ fs::path_ext(.x))

  # Create a frequency count of file extensions
  fileTypeCounts <- fileExt %>%
    tibble::as_tibble() %>%
    dplyr::group_by(value) %>%
```

```
24      dplyr::tally(sort = TRUE) %>%
```

```
You're right, creating a regex pattern to match the specified file types can be a more
elegant solution. The function fs::dir_ls() can accept a regex pattern in its glob argument.
```





```
fileTypeCounts('example_files', fileTypes=c('csv', 'json'))
```

```
## # A tibble: 0 × 2
## # ℹ 2 variables: File_Type <chr>, Count <int>
```

ChatGPT

That returns 0 results. I think the glob.pattern is wrong and that you should use the regex argument, not the glob argument.





```r
fileTypeCounts <- function(dirPath, fileTypes = NULL) {
  # Ensure the directory exists
  if(!fs::dir_exists(dirPath)) {
    stop("Directory not found")
  }

  # If specific file types are provided, modify the regex pattern accordingly
  if (!is.null(fileTypes)) {
    regex_pattern <- paste0(".*\\.(", paste(fileTypes, collapse = "|"), ")$")
  } else {
    regex_pattern <- ".*"  # Matches any file
  }

  # List files in the directory according to the regex pattern
  fileList <- fs::dir_ls(dirPath, regexp = regex_pattern)

  # Get file extensions
  fileExt <- purrr::map_chr(fileList, ~ fs::path_ext(.x))

  # Create a frequency count of file extensions
  fileTypeCounts <- fileExt %>%
    tibble::as_tibble() %>%
    dplyr::group_by(value) %>%
    dplyr::tally(sort = TRUE) %>%
    dplyr::rename(File_Type = value, Count = n)
```





```
fileTypeCounts('example_files', fileTypes=c('csv', 'json'))
```

```
## # A tibble: 2 × 2
##   File_Type Count
##   <chr>     <int>
## 1 csv           3
## 2 json          1
```

ChatGPT

Can you write the roxygen documentation for this function?

```r
#' Count File Types in Directory
#'
#' This function counts the number of each file type in a specified directory.
#' The file type is determined by the file extension. If specific file types are
#' provided, the function only counts files of those types.
#'
#' @param dirPath Character string specifying the path to the directory.
#' @param fileTypes Optional character vector specifying which file types to consider.
#'   If NULL (the default), all file types are considered.
#'
#' @return A tibble with one row for each file type and columns for the file type and cou
#'   The counts are sorted in descending order.
#'
#' @examples
#' \dontrun{
#' fileTypeCounts("/path/to/folder", c("txt", "csv"))
#' }
#'
#' @export
fileTypeCounts <- function(dirPath, fileTypes = NULL) {
  # Ensure the directory exists
  if(!fs::dir_exists(dirPath)) {
    stop("Directory not found")
  }
```

```
The \dontrun{} around the example is used because the example includes a directory path that
won't exist on the user's system. This prevents R from trying to run the example when
checking the package.
```

ChatGPT

Make a similar function, that calculates the number of lines of each function and returns the average for each file type. For file types where the number of lines cannot be computed, that should return NA. It should have the same arguments as this function and allow us to specify certain file types. It should use the tidyverse and package::function() notation.

```r
avgLinesPerFile <- function(dirPath, fileTypes = NULL) {
  # Ensure the directory exists
  if(!fs::dir_exists(dirPath)) {
    stop("Directory not found")
  }

  # If specific file types are provided, modify the regex pattern accordingly
  if (!is.null(fileTypes)) {
    regex_pattern <- paste0(".*\\.(", paste(fileTypes, collapse = "|"), ")$")
  } else {
    regex_pattern <- ".*"  # Matches any file
  }

  # List files in the directory according to the regex pattern
  fileList <- fs::dir_ls(dirPath, regexp = regex_pattern)

  # Get file extensions
  fileExt <- purrr::map_chr(fileList, ~ fs::path_ext(.x))

  # Calculate the number of lines for each file (returns NA for binary files)
```

```
21  numLines <- purrr::map_dbl(fileList, ~ tryCatch(length(readLines(.x)), error = functior
22
23  # Combine file extensions and number of lines into a tibble
24  fileData <- tibble::tibble(File_Type = fileExt, Num_Lines = numLines)
```

```
avgLinesPerFile('example_files')
```

```
## # A tibble: 3 × 2
##   File_Type Avg_Lines
##   <chr>         <dbl>
## 1 csv            6.67
## 2 json          42
## 3 txt            8
```

```
avgLinesPerFile('example_files', fileTypes=c('csv', 'json'))
```

```
## # A tibble: 2 × 2
##   File_Type Avg_Lines
##   <chr>         <dbl>
## 1 csv            6.67
## 2 json          42
```





ChatGPT

Now write a function that uses this function in order generate a bar plot of the average number of lines for each file type.





```r
#' Bar Plot of Average Lines Per File by File Type
#'
#' This function generates a bar plot of the average number of lines for each file type in
#' The number of lines is only calculated for file types that are text-based (e.g., .txt, .
#' For binary file types, the number of lines cannot be computed and the function returns N
#'
#' @export
plotAvgLinesPerFile <- function(dirPath, fileTypes = NULL) {
  # Calculate average lines per file by file type
  avgLines <- avgLinesPerFile(dirPath, fileTypes)

  # Generate the bar plot
  plot <- ggplot2::ggplot(avgLines, ggplot2::aes(x = File_Type, y = Avg_Lines)) +
    ggplot2::geom_bar(stat = "identity", fill = "skyblue") +
    ggplot2::theme_minimal() +
    ggplot2::theme(axis.text.x = ggplot2::element_text(angle = 45, hjust = 1)) +
    ggplot2::labs(x = "File Type", y = "Average Number of Lines",
                  title = "Average Number of Lines per File by File Type")

  return(plot)
}
```





```
1 plotAvgLinesPerFile('example_files') |> print()
```

ChatGPT

Modify the function to allow users to choose either ggplot or echarts and then make whichever version the user requested.

```r
plotAvgLinesPerFile <- function(dirPath, fileTypes = NULL, plotLibrary = "ggplot") {
  # Calculate average lines per file by file type
  avgLines <- avgLinesPerFile(dirPath, fileTypes)

  # Check the plot library argument
  if (!plotLibrary %in% c("ggplot", "echarts")) {
    stop("Invalid plot library. Choose either 'ggplot' or 'echarts'.")
  }

  # Generate the bar plot with the specified library
  if (plotLibrary == "ggplot") {
    # Generate the ggplot bar plot
    plot <- ggplot2::ggplot(avgLines, ggplot2::aes(x = File_Type, y = Avg_Lines)) +
      ggplot2::geom_bar(stat = "identity", fill = "skyblue") +
      ggplot2::theme_minimal() +
      ggplot2::theme(axis.text.x = ggplot2::element_text(angle = 45, hjust = 1)) +
      ggplot2::labs(x = "File Type", y = "Average Number of Lines",
                    title = "Average Number of Lines per File by File Type")
  } else {
    # Generate the echarts bar plot
    plot <- echartr::echartr(avgLines, File_Type, Avg_Lines, "bar") %>%
      echartr::setToolbox(show = TRUE) %>%
      echartr::setTitle("Average Number of Lines per File by File Type") %>%
      echartr::setxAxis(name = "File Type") %>%
```

ChatGPT

It looks like you’re trying to use echartr rather than echarts4r.

```r
plotAvgLinesPerFile <- function(dirPath, fileTypes = NULL, plotLibrary = "ggplot") {
  # Calculate average lines per file by file type
  avgLines <- avgLinesPerFile(dirPath, fileTypes)

  # Check the plot library argument
  if (!plotLibrary %in% c("ggplot", "echarts4r")) {
    stop("Invalid plot library. Choose either 'ggplot' or 'echarts4r'.")
  }

  # Generate the bar plot with the specified library
  if (plotLibrary == "ggplot") {
    # Generate the ggplot bar plot
    plot <- ggplot2::ggplot(avgLines, ggplot2::aes(x = File_Type, y = Avg_Lines)) +
      ggplot2::geom_bar(stat = "identity", fill = "skyblue") +
      ggplot2::theme_minimal() +
      ggplot2::theme(axis.text.x = ggplot2::element_text(angle = 45, hjust = 1)) +
      ggplot2::labs(x = "File Type", y = "Average Number of Lines",
                    title = "Average Number of Lines per File by File Type")
  } else {
    # Generate the echarts4r bar plot
    plot <- avgLines %>%
      echarts4r::e_charts(File_Type) %>%
      echarts4r::e_bar(Avg_Lines) %>%
      echarts4r::e_title("Average Number of Lines per File by File Type")
```

```
1 plotAvgLinesPerFile('example_files', plotLibrary='echarts4r')
```



ChatGPT

Instead of checking for allowed plot libraries in an if statement, make the default to the argument a vector of allowable types then check with match.arg().

```r
plotAvgLinesPerFile <- function(dirPath, fileTypes = NULL, plotLibrary = c("ggplot", "ech
  # Calculate average lines per file by file type
  avgLines <- avgLinesPerFile(dirPath, fileTypes)

  # Match the plot library argument
  plotLibrary <- match.arg(plotLibrary)

  # Generate the bar plot with the specified library
  if (plotLibrary == "ggplot") {
    # Generate the ggplot bar plot
    plot <- ggplot2::ggplot(avgLines, ggplot2::aes(x = File_Type, y = Avg_Lines)) +
      ggplot2::geom_bar(stat = "identity", fill = "skyblue") +
      ggplot2::theme_minimal() +
      ggplot2::theme(axis.text.x = ggplot2::element_text(angle = 45, hjust = 1)) +
      ggplot2::labs(x = "File Type", y = "Average Number of Lines",
                    title = "Average Number of Lines per File by File Type")
  } else {
    # Generate the echarts4r bar plot
    plot <- avgLines %>%
      echarts4r::e_charts(File_Type) %>%
      echarts4r::e_bar(Avg_Lines) %>%
      echarts4r::e_title("Average Number of Lines per File by File Type")
  }
```



ChatGPT

Rather than put all the plotting code in this function, put each type of plot in its own helper function, then have this function call one of those based on what the user specified.

```r
#' Generate ggplot Bar Plot
#'
#' @param data Dataframe containing the average number of lines and file types.
#' @return ggplot object.
genGgplot <- function(data) {
  plot <- ggplot2::ggplot(data, ggplot2::aes(x = File_Type, y = Avg_Lines)) +
    ggplot2::geom_bar(stat = "identity", fill = "skyblue") +
    ggplot2::theme_minimal() +
    ggplot2::theme(axis.text.x = ggplot2::element_text(angle = 45, hjust = 1)) +
    ggplot2::labs(x = "File Type", y = "Average Number of Lines",
                  title = "Average Number of Lines per File by File Type")
  return(plot)
}

#' Generate echarts4r Bar Plot
#'
#' @param data Dataframe containing the average number of lines and file types.
#' @return echarts4r plot object.
genEcharts <- function(data) {
  plot <- data %>%
    echarts4r::e_charts(File_Type) %>%
    echarts4r::e_bar(Avg_Lines) %>%
    echarts4r::e_title("Average Number of Lines per File by File Type")
```





```
24    return(plot)
```

```r
plotAvgLinesPerFile <- function(dirPath, fileTypes = NULL, plotLibrary = c("ggplot", "echar
  # Calculate average lines per file by file type
  avgLines <- avgLinesPerFile(dirPath, fileTypes)

  # Match the plot library argument
  plotLibrary <- match.arg(plotLibrary)

  # Generate the bar plot with the specified library
  if (plotLibrary == "ggplot") {
    plot <- genGgplot(avgLines)
  } else {
    plot <- genEcharts(avgLines)
  }

  return(plot)
}
```





ChatGPT

Almost got it, but use match.arg() here too.

```
plotAvgLinesPerFile <- function(dirPath, fileTypes = NULL, plotLibrary = c("ggplot", "echar
  # Calculate average lines per file by file type
  avgLines <- avgLinesPerFile(dirPath, fileTypes)

  # Match the plot library argument and create a list of function calls
  plotFunc <- list(
    ggplot = genGgplot(avgLines),
    echarts4r = genEcharts(avgLines)
  )

  # Generate the bar plot with the specified library
  plot <- plotFunc[[match.arg(plotLibrary)]]

  return(plot)
}
```

ChatGPT

Add options for a base R plot and lattice.

```r
#' Generate Base R Bar Plot
#'
#' @param data Dataframe containing the average number of lines and file types.
#' @return NULL. The function outputs the plot directly.
genBaseRPlot <- function(data) {
  barplot(data$Avg_Lines, names.arg = data$File_Type,
          main = "Average Number of Lines per File by File Type",
          xlab = "File Type", ylab = "Average Number of Lines", col = "skyblue")
}

#' Generate Lattice Bar Plot
#'
#' @param data Dataframe containing the average number of lines and file types.
#' @return lattice plot object.
genLatticePlot <- function(data) {
  plot <- lattice::barchart(Avg_Lines ~ File_Type, data = data,
                            main = "Average Number of Lines per File by File Type",
                            xlab = "File Type", ylab = "Average Number of Lines",
                            col = "skyblue")
  return(plot)
}

#' Bar Plot of Average Lines Per File by File Type
#'
```

```
1  plotAvgLinesPerFile('example_files', plotLibrary='base') |> print()
```

```
##      [,1]
## [1,]  0.7
## [2,]  1.9
## [3,]  3.1
```

```
plotAvgLinesPerFile('example_files', plotLibrary='lattice') |> print()
```

ChatGPT

It looks like this creates a bug where the lattice plots are plotted twice. So instead of making a list of generated plots (this is wasteful anyway), only call the helper function for the one plot type chosen.

```r
plotAvgLinesPerFile <- function(dirPath, fileTypes = NULL, plotLibrary = c("ggplot", "echar
  # Calculate average lines per file by file type
  avgLines <- avgLinesPerFile(dirPath, fileTypes)

  # Match the plot library argument and create a list of function calls
  plotFunc <- list(
    ggplot = genGgplot,
    echarts4r = genEcharts,
    base = genBaseRPlot,
    lattice = genLatticePlot
  )

  # Generate the bar plot with the specified library
  plot <- plotFunc[[match.arg(plotLibrary)]](avgLines)

  return(plot)
}
```





```
plotAvgLinesPerFile('example_files', plotLibrary='lattice') |> print()
```

ChatGPT

I want to turn these functions into a package. Can you outline the steps needed to do that?

Sure, here are the general steps you would follow to create a package in R:

1. **Setup:** Install the `devtools` and `roxygen2` packages, which provide functions to easily create and document your package.

    ```r
    install.packages(c("devtools", "roxygen2"))
    ```

2. **Create the package structure:** Use `devtools::create("yourPackageName")` to create the package structure. This will create a new directory in your current working directory named "yourPackageName", with the necessary files and directories for an R package.

3. **Add your functions:** Place your R script files containing your functions into the "R" subdirectory of your package directory. If your functions are currently in a script in your working directory, you can use `file.copy()` to copy them into the "R" subdirectory.

4. **Document your functions:** Use roxygen comments (`#'`) to document each function. You've already done this step for your functions. The roxygen comments should be placed immediately above each function in your R scripts.

ChatGPT

Can you help me come up with a name for this package?

```
Sure, naming a package can be a fun and creative process. The main goal is to create a
name that is memorable, fairly unique, and gives some hint as to what the package does.
Here are a few suggestions based on the functionality of your package:

1. `fileTypeMetrics`: A straightforward name that conveys the package's main
functionality. The package provides metrics (e.g., average number of lines) for different
file types.

2. `lineCounter`: This name focuses on the action of counting lines in different file
types. It's slightly more abstract but still relevant.

3. `fileTypeAnalyzer`: A more generic name that could allow for additional features in the
future, related to analyzing different file types.

4. `codeLineStats`: If your primary focus is code files (e.g., .R, .py), this name
emphasizes that focus.

5. `fileStatPlot`: This name highlights both the statistical analysis and the plotting
features of your package.

Remember, it's also a good idea to search for your intended name on CRAN and Google to
```





ChatGPT

Let’s go with fileTypeAnalyzer.

````
That's a great choice! `fileTypeAnalyzer` is a descriptive and easy-to-understand name
that suggests the primary functionality of your package. Here are the next steps to
initialize your package:

1. **Setup:** Make sure the `devtools` and `roxygen2` packages are installed. If not,
install them using `install.packages(c("devtools", "roxygen2"))`.

2. **Create the package structure:** Now, use the `devtools` package to create the
structure of your package. Open your R console, set your working directory to where you
want the package to be created, and use the `create` function from `devtools`:

    ```r
    setwd("/path/to/your/desired/directory")
    devtools::create("fileTypeAnalyzer")
    ```

This will create a new directory in your current working directory named
"fileTypeAnalyzer", with the necessary files and directories for an R package.

3. **Add your functions:** Now, you can start adding your functions to the "R"
subdirectory of your package directory. Make sure to document them as we discussed before,
````

ChatGPT

Can you come up with what should go in the “DESCRIPTION” field?

```
Description: The 'fileTypeAnalyzer' package provides tools for analyzing files
    in a directory by their type. It calculates metrics such as the average number
    of lines for each file type and can create plots of these metrics.
    Currently supported file types for analysis include but are not limited to
    'txt', 'csv', 'R', etc. Custom file types can also be specified by the user.
License: GPL-3
```





DESCRIPTION

```
Package: fileTypeAnalyzer
Title: Analyzing Files by Their Type
Version: 0.1.0
Authors@R:
    person("Your", "Name", email = "your.email@example.com", role = c("aut", "cre"))
Description: The 'fileTypeAnalyzer' package provides tools for analyzing files
    in a directory by their type. It calculates metrics such as the average number
    of lines for each file type and can create plots of these metrics.
    Currently supported file types for analysis include but are not limited to
    'txt', 'csv', 'R', etc. Custom file types can also be specified by the user.
License: GPL-3
Encoding: UTF-8
LazyData: true
Depends:
    R (>= 3.5.0)
Imports:
    dplyr,
    purrr,
    fs,
    readr,
    stringr,
```





ChatGPT

Let’s use the MIT licesne.

```
Sure, you can change the License field to MIT + file LICENSE to specify the MIT license.

In addition to this, you need to add a LICENSE file in your package root directory with the
full text of the MIT license. This is a requirement of the MIT license itself.
```

LICENSE

```
MIT License

Copyright (c) [year] [fullname]

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```





ChatGPT

Can you write tests for each of the functions included in the package?

```
Absolutely, testing is an important part of package development. The testthat package is
widely used in R for writing unit tests. Here are some examples of tests for your functions.

First, you will need to structure your tests appropriately. testthat expects tests to be in
files within a tests/testthat/ directory in your package.
```

For instance, your tests for the count_file_types function might look something like this, placed in a file called test-count_file_types.R:

```r
context("Test count_file_types function")

# Load the necessary packages
library(testthat)
library(fileTypeAnalyzer)

# Test if function correctly counts file types
test_that("count_file_types correctly counts file types", {
  # Assume we have a test directory "test_dir" with the files
  # "file1.txt", "file2.txt", "file3.csv" and "file4.R"

  expect_equal(
    count_file_types("test_dir", file_types = c("txt", "csv", "R")),
    data.frame(file_type = c("txt", "csv", "R"),
               count = c(2, 1, 1))
  )
})
```

Similarly, you can write tests for the average_lines_by_file_type function in a file called test-average_lines_by_file_type.R:

```r
context("Test average_lines_by_file_type function")

# Load the necessary packages
library(testthat)
library(fileTypeAnalyzer)

# Test if function correctly calculates average lines
test_that("average_lines_by_file_type correctly calculates average lines", {
  # Assume we have a test directory "test_dir" with the files
  # "file1.txt" (10 lines), "file2.txt" (20 lines), "file3.csv" (15 lines)
  # and "file4.R" (25 lines)

  expect_equal(
    average_lines_by_file_type("test_dir", file_types = c("txt", "csv", "R")),
    data.frame(file_type = c("txt", "csv", "R"),
               avg_lines = c(15, 15, 25))
  )
})
```

```
For the plot_avg_lines function, you could write a test to check that the output is a plot
object. However, this test would be less straightforward, since ggplot objects, echarts, and
lattice and base R plots don't have the same class or structure. But for a basic test, you
could just check if a plot is returned (i.e., no error is thrown) when you call the
function:
```





```r
context("Test plot_avg_lines function")

# Load the necessary packages
library(testthat)
library(fileTypeAnalyzer)

# Test if function correctly produces a plot
test_that("plot_avg_lines correctly produces a plot", {
  # Assume we have a test directory "test_dir" with the files
  # "file1.txt" (10 lines), "file2.txt" (20 lines), "file3.csv" (15 lines)
  # and "file4.R" (25 lines)

  expect_silent(
    plot_avg_lines("test_dir", file_types = c("txt", "csv", "R"), plot_lib = "ggplot")
  )

  expect_silent(
    plot_avg_lines("test_dir", file_types = c("txt", "csv", "R"), plot_lib = "echarts4r")
  )

  expect_silent(
    plot_avg_lines("test_dir", file_types = c("txt", "csv", "R"), plot_lib = "base")
  )

```

```
To run the tests, you can use devtools::test() or testthat::test_dir("tests/testthat") from
your package root directory.

Note that these tests assume that you have a directory "test_dir" with the files
"file1.txt", "file2.txt", "file3.csv" and "file4.R". In real tests, you should set up and
tear down these files using testthat::setup() and testthat::teardown(), or consider using
the withr package to temporarily change the working directory during tests.

Also note that tests that compare against hard-coded expected output (like our tests for
count_file_types and average_lines_by_file_type) can be brittle, especially if the output is
subject to small changes or if the function behavior changes slightly. In practice, you
would likely write multiple, more granular tests, and possibly use
testthat::expect_equal_to_reference() or similar functions to avoid hard-coding expected
output in your tests.
```





ChatGPT

How would I use testthat::setup() and testthat::teardown()?

The setup() and teardown() functions from testthat are used to prepare the testing environment before the tests are run and to clean up afterwards. They are often used to create and delete temporary files or directories, set and reset options, etc.

```
context("Test count_file_types function")

# Setup: Create a temporary directory and some files
temp_dir <- tempfile()
dir.create(temp_dir)
file.create(file.path(temp_dir, "file1.txt"))
file.create(file.path(temp_dir, "file2.txt"))
file.create(file.path(temp_dir, "file3.csv"))
file.create(file.path(temp_dir, "file4.R"))

# Register a teardown function to remove the temporary directory and files
teardown_env <- new.env(parent = emptyenv())
setup({
  assign("temp_dir", temp_dir, envir = teardown_env)
})
teardown({
  unlink(get("temp_dir", envir = teardown_env), recursive = TRUE)
})

# Test if function correctly counts file types
test_that("count_file_types correctly counts file types", {
```

```
22    expect_equal(
23      count_file_types(temp_dir, file_types = c("txt", "csv")),
24      data.frame(file_type = c("txt", "csv"),
```

ChatGPT

Should we populate data into those temporary sample files?

```r
context("Test average_lines_by_file_type function")

# Setup: Create a temporary directory and some files
temp_dir <- tempfile()
dir.create(temp_dir)

# create text files with different number of lines
writeLines(rep("Line of text", 5), file.path(temp_dir, "file1.txt"))
writeLines(rep("Line of text", 10), file.path(temp_dir, "file2.txt"))
writeLines(rep("Line of text", 15), file.path(temp_dir, "file3.txt"))
writeLines(rep("Line of text", 20), file.path(temp_dir, "file4.txt"))


# Test if function correctly calculates average lines
test_that("average_lines_by_file_type correctly calculates average lines", {
  expect_equal(
    average_lines_by_file_type(temp_dir, file_types = c("txt")),
    data.frame(file_type = "txt", avg_lines = 12.5)
  )
})
```

ChatGPT

Do we have to run the setup and teardown for each test file? Or can we do it once for all of them?

```
The setup and teardown code is generally run for each test file, as each file might require
a different setup and teardown. testthat runs each test file in a new environment, so any
objects created in one test file will not be available in another. This isolation helps to
ensure that tests do not interfere with each other.

However, if you have some setup and teardown code that needs to be run for all test files,
there are a couple of ways you could handle this in testthat.

...

Remember that tests should ideally be independent from each other, so be careful not to
introduce dependencies between tests with your setup and teardown code. If tests are not
independent, a failure in one test could cause failures in other tests, making the test
results harder to interpret.
```





ChatGPT

For the plotting functions, can you test using a visual differ?

```
Visual regression testing, sometimes called "visual diffing", can be used to check if the
visual output of a function remains consistent over time. In the context of R programming
and testing R packages, the vdiffr package is commonly used for this purpose.

The vdiffr package allows you to create visual unit tests for your functions that generate
ggplot2, base, or lattice plots. When you run the tests, vdiffr will compare the current
output of your plotting function to a saved version of the output. If the plots are not
identical, vdiffr will flag the test as failing.

To implement visual regression tests for your plotting functions, you would first add a
dependency on vdiffr in your DESCRIPTION file:
```

```r
context("Test plot_avg_lines function")

library(testthat)
library(vdiffr)
library(fileTypeAnalyzer)

test_that("plot_avg_lines creates the correct plot", {
  temp_dir <- tempdir()
  # Create a data frame of average line counts by file type
  avg_lines <- data.frame(
    file_type = c("txt", "csv", "R"),
    avg_lines = c(10, 20, 30)
  )

  # Test ggplot version
  plot <- plot_avg_lines(avg_lines, "ggplot")
  expect_doppelganger("Correct ggplot", plot)

  # Test base version
  plot <- plot_avg_lines(avg_lines, "base")
  expect_doppelganger("Correct base plot", plot)

  # Test lattice version
  plot <- plot_avg_lines(avg_lines, "lattice")
  expect_doppelganger("Correct lattice plot", plot)
```

```
devtools::test()
```

```
ℹ Testing fileTypeAnalyzer
✓ | F W S  OK | Context
✓ |         1 | Test average_lines_by_file_type function
✓ |         1 | Test count_file_types function
✓ |         8 | Test plot_avg_lines function

══ Results ═════════════════════════════════════════════════════════════════════
[ FAIL 0 | WARN 0 | SKIP 0 | PASS 10 ]
```





```
devtools::check()
```

```
── R CMD check results ─────────────────────────────────────────────────────────
fileTypeAnalyzer 0.1.0 ────
Duration: 9.4s

❯ checking dependencies in R code ... WARNING
  '::' or ':::' import not declared from: ‘tibble’
  Namespaces in Imports field not imported from:
    ‘readr’ ‘stringr’
    All declared Imports should be used.

❯ checking R code for possible problems ... NOTE
  avgLinesPerFile: no visible global function definition for ‘%>%’
  avgLinesPerFile: no visible binding for global variable ‘File_Type’
  avgLinesPerFile: no visible binding for global variable ‘Num_Lines’
  fileTypeCounts: no visible global function definition for ‘%>%’
  fileTypeCounts: no visible binding for global variable ‘value’
  fileTypeCounts: no visible binding for global variable ‘n’
  genBaseRPlot: no visible global function definition for ‘barplot’
  genEcharts: no visible global function definition for ‘%>%’
  genEcharts: no visible binding for global variable ‘File_Type’
  genEcharts: no visible binding for global variable ‘Avg_Lines’
```

ChatGPT

Draft a tweet announcing the package.

```
Sure, here's a suggestion for a tweet announcing your new package:

"🚀 Exciting news in the #rstats world! We just released fileTypeAnalyzer, a new #rpackage
that makes it easy to analyze your file types and contents. Explore different file types,
count lines, and visualize your data. Get started today at https://github.com/your-github-
username/fileTypeAnalyzer #DataScience"

Remember to replace the GitHub URL with the actual URL of your package. If you have any
screenshots or images (like your new hex sticker), those would be great to include in your
tweet as well.
```





DALL-E

A calculator superimposed over a stack of papers in the style of Andy Warhol.



ChatGPT

How do I convert an image into a hex sticker?

```
library(hexSticker)
sticker(
    here::here('images', 'dall-e-hex-sticker.png'),
    package = "fileTypeAnalyzer",  # your package name
    p_size = 14,  # size of the package text
    p_color = "blue",  # color of the package text
    p_y = 0.6,  # position of the package text
    s_x = 1,  # x position of the sticker
    s_y = 1,  # y position of the sticker
    s_width = 1,  # width of the sticker
    s_height = 1,  # height of the sticker
    h_fill='white',
    filename = here::here('images', 'fileTypeAnalyzer_hex.png')  # output filename,
)
```









What Did We Learn?





- LLM can write decent functions quickly
- Not great functions, but good and fast
- Made a bunch of mistakes that I needed to catch
- Needs guidance on style
- Needs direction on less used tools
- Get confused the further you go
- Needs a human checking the work





https://github.com/jaredlander/fileTypeAnalyzer





https://chat.openai.com/share/1d2b4efd-42d4-483d-b593-11363e395415

Thank You








lander
analytics