

Introduction to R and Python

Magno Severino

2024-03-03

Table of contents

Introduction	5
I Learning R	6
1 R Fundamentals	7
1.1 RStudio	7
1.2 Data Types	8
1.2.1 Numeric Data Type	8
1.2.2 Logical Data Type	9
1.2.3 Character Data Type	9
1.2.4 Factor Data Type	10
1.3 R language fundamentals	10
1.4 Variables	11
1.5 Checking the Type of a Variable	11
1.6 Data structures	13
1.6.1 Vectors	13
1.6.2 Matrices	15
1.6.3 Lists	17
1.6.4 :::	20
1.6.5 DataFrames	20
1.7 Exercises	22
2 Execution Flows	23
2.1 Conditional Structures	23
2.1.1 <code>if</code> and <code>else</code>	23
2.1.2 <code>else if</code>	24
2.2 Loop Structures	25
2.2.1 <code>for</code>	25
2.2.2 <code>while</code>	28
2.3 Functions	31
2.4 Packages	33
2.5 Exercises	33
3 Data Manipulation	35
3.1 Importing External Files	35

3.2	The tidyverse Package	36
3.3	The > Pipe Operator	36
3.4	Data in <i>Tidy</i> Format	37
3.5	Main Verbs of the dplyr Package	40
3.5.1	select	41
3.5.2	arrange	41
3.5.3	filter	43
3.5.4	mutate	44
3.5.5	summarise	44
3.5.6	group by	45
3.6	Helper Functions	46
3.7	Exercises	50
4	Data Visualization	53
4.1	Grammar of Graphics	53
4.2	The ggplot Package	53
4.2.1	Data	53
4.2.2	Aesthetics	55
4.2.3	Geometry	56
4.2.4	Facets	62
4.2.5	Coordinates	63
4.2.6	Themes	65
4.2.7	Customization and Styling of Graphs	67
4.3	Additional Packages	71
4.3.1	The patchwork Package	71
4.3.2	The ggthemes Package	73
4.3.3	The plotly Package	74
4.4	Extra Tips	75
4.5	Exercises	75
II	Learning Python	79
5	Fundamentos de Python	80
5.1	Instalação	80
5.2	Tipos de dados fundamentais	81
5.2.1	O tipo de dado inteiro	81
5.2.2	O tipo de dado ponto flutuante	82
5.2.3	O tipo de dado cadeia de caracteres	82
5.2.4	O tipo de dado lógico	83
5.2.5	Coerção de tipos	84
5.3	Objetos básicos	85
5.3.1	Listas	85

5.3.2	Tuplas	87
5.3.3	Dicionários	88
5.4	Fatias (<i>slices</i>)	89
5.5	Condicionais	91
5.6	Estruturas repetitivas	91
5.6.1	<code>for</code>	92
5.6.2	<code>while</code>	93
5.7	Comprehensions	93
5.8	Funções	95
5.8.1	Função <code>lambda</code>	96
5.9	Classes e objetos	96
5.10	Exercícios	98
6	Processamento e visualização de dados	102
6.1	Instalação de bibliotecas	102
6.2	Processamento de dados numéricos	102
6.3	Análise e processamento de dados	106
6.4	O que é pandas?	107
6.4.1	Séries	107
6.4.2	DataFrame	108
6.4.3	Principais funcionalidades	109
6.5	Dados organizados (tidy data)	114
6.6	Visualização de dados	119
6.6.1	Matplotlib	119
6.6.2	Plotnine	122
III	Case studies	124
	References	125

Introduction

This book consists of lecture notes from the short course *Introduction to Programming Languages R and Python*.

If you are interested in the fundamentals of the R language, you can find these topics in the first part of the book. On the other hand, if you want to explore the fundamentals of Python, the second part is where you will find that information.

This is a continuously evolving book, designed to grow and adapt. If you have any suggestions or feedback on the content presented, feel free to send an email to [magno-tairone\[at\]gmail.com](mailto:magno-tairone[at]gmail.com).

My sincere thanks to Luiza Tuler for reviewing the content of this book.

Part I

Learning R

1 R Fundamentals

1.1 RStudio

To install R, download it from <http://www.r-project.org>. Then, install the IDE (Integrated Development Environment) **R Studio**.

When you open RStudio, click on the *File/ New File/ R Script* menu (or press Ctrl+Shift+N). You should see a structure similar to the one shown in the figure below.

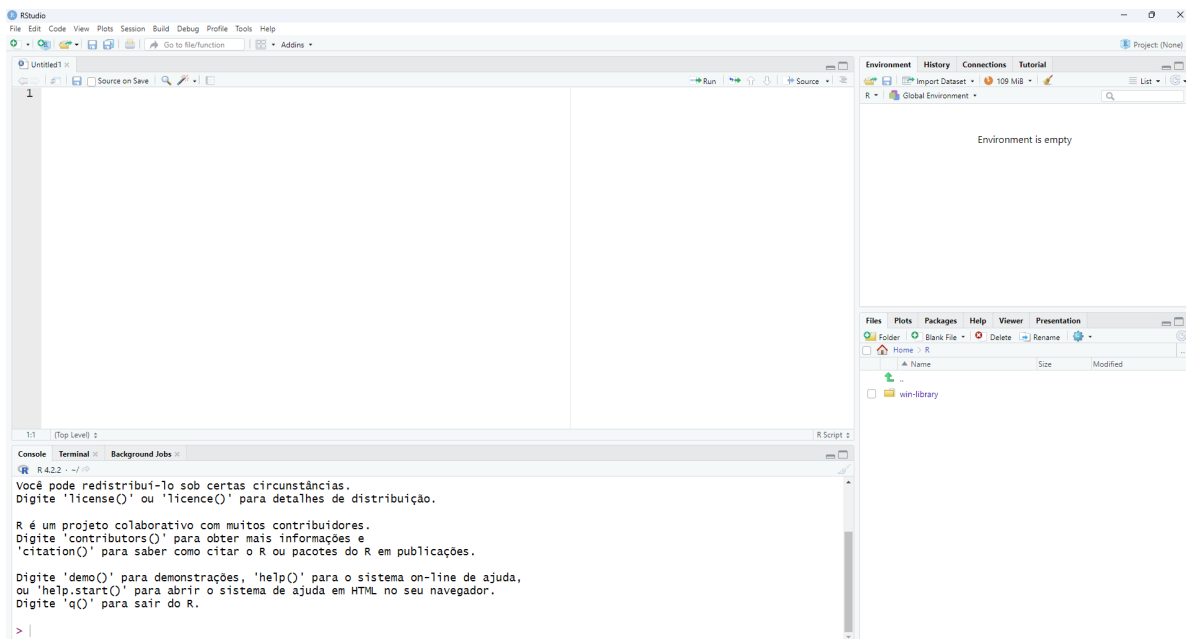


Figure 1.1: RStudio Interface

Note that there are four panels:

- *Script Panel* (top left): This panel is where you can write, edit, and run R scripts. It provides features such as syntax highlighting, autocomplete, and code checking to assist with coding.

- *Console Panel* (bottom left): The console is where R code is executed and results are displayed. You can enter commands directly here and see the results immediately. It also maintains a history of executed commands, which can be useful for future reference.
- *Environment/Workspace* (top right): This panel displays information about objects (such as variables, functions, etc.) currently loaded in R's memory. It shows details such as the object's name, type, and current value. This is useful for monitoring and managing objects during a work session.
- *Files/Plots/Packages/Help* (bottom right): A panel with various features.
 - Files: This tab allows you to browse and manage your project's files. You can create, rename, delete, and organize files and folders directly within RStudio.
 - Plots: Here, the plots generated by R are displayed. When you create a plot using visualization functions in R, the result is shown in this tab. This facilitates visual data analysis and inspection during the creation process.
 - Packages: In this tab, you can view and manage the packages installed in your R environment. It displays a list of all installed packages, along with their version and status (loaded or not). Additionally, you can install new packages, update existing ones, and load or unload packages as needed for your work.
 - Help: This tab provides quick access to documentation and help information about R functions, packages, and other resources. You can search for specific topics and access the official documentation directly within RStudio. This is useful for obtaining information about a function's syntax, usage examples, and details about available parameters.

1.2 Data Types

Whenever you learn a new programming language, it's essential to first understand the basic data types it supports.

In R, there are four basic data types available: numeric, logical, character, and factor.

1.2.1 Numeric Data Type

Numeric data is used to express quantitative values, such as prices, rates, and quantities, and is represented by integers or decimals.

```
# Integer representing the number of shares in a portfolio
num_shares <- 100
```



```
# Floating-point number representing the inflation rate
inflation_rate <- 3.5

# Checking the class of inflation_rate
class(inflation_rate)
```

```
[1] "numeric"
```

The `class()` function is used to determine the class of a variable. In other words, it provides information about the type of data that a variable represents. In the example above, the `inflation_rate` variable belongs to the `numeric` class.

1.2.2 Logical Data Type

Logical data is used to represent states or conditions, such as true or false, and is useful in logical operations and comparisons.

```
# Checking if the interest rate is increasing
interest_rate_rising <- TRUE

# Checking if stock prices are falling
stock_price_falling <- FALSE

# Checking the class of stock_price_falling
class(stock_price_falling)
```

```
[1] "logical"
```

1.2.3 Character Data Type

Character data is used to represent text, such as country names, company names, or categories, and is essential for descriptive analyses and communication of results.

```
# Name of a country
country <- "Brazil"

# Name of a multinational company
company <- "Petróleo Brasileiro S.A."
```

```
# Checking the class of country
class(country)
```

```
[1] "character"
```

1.2.4 Factor Data Type

Factors are used to represent categorical variables, such as classifications, categories, or groups, offering an efficient way to handle discrete and qualitative data.

```
# Credit risk classification of a company
credit_risk <- factor(c("Low", "Medium", "High", "Low", "High"))

# Checking the class of credit_risk
class(credit_risk)
```

```
[1] "factor"
```

The `levels()` function returns the levels (or categories) of a factor. This is useful to understand which categories are represented and to perform data manipulation operations based on these categories.

```
# displaying credit risk levels
levels(credit_risk)
```

```
[1] "High"    "Low"     "Medium"
```

1.3 R language fundamentals

The R environment refers to the workspace where all variables, functions, and objects created during an R session are stored and managed. The environment includes both the objects you create and those that are automatically loaded via packages or other data import mechanisms (more on packages in [Section 2.4](#)).

For example, using the `ls()` function (which lists the names of objects in the current environment), we can see all objects currently present in the R environment.

```
ls()
```

If you have correctly executed all the commands in Section 1.2, you should see the following output in the console:

```
[1] "company"           "country"           "num_shares"
[4] "stock_price_falling" "inflation_rate"    "interest_rate_rising"
```

1.4 Variables

In Section 1.2, several variables were created. For example, the `company` variable stores a character string. Previously, you saw how to list all defined variables in your environment. But what exactly are variables?

In R, variables are fundamental elements used to store and manipulate data. They are like containers that hold values, objects, or expressions. When you assign a value to a variable, you are essentially giving it a name to access and manipulate it later.

For example, by writing `stock_price <- 10`, you are creating a variable called `stock_price` and assigning it the value 10. Now, whenever you use `stock_price` in your code, you are referring to this value.

It is a common practice to choose descriptive variable names that help understand their purpose or content. For example, in an economic context, you might use `stock_price` to represent a stock's price or `inflation_rate` to represent the inflation rate.

To assign a value to a variable, use the `<-` operator. The `=` operator can also be used for assignment. Both have the same practical effect in R, and the choice between them usually comes down to personal preference and coding style, although some style guides recommend using `<-`.

1.5 Checking the Type of a Variable

We can use the `is.*` family of functions to verify the types of some variables in our workspace.

- For the `company` variable:

```
is.character(company)
```

This will return `TRUE` if the `company` variable is of type character.

- For the `country` variable:

```
is.character(country)
```

Just like for the variable `company`, this will return `TRUE` if the variable `country` is of type character.

- For the variable `num_shares`:

```
is.numeric(num_shares)
```

This will return `TRUE` if the variable `num_shares` is of type numeric.

- For the variable `stock_price_falling`:

```
is.logical(stock_price_falling)
```

This will return `TRUE` if the variable `stock_price_falling` is of type logical.

- For the variable `inflation_rate`:

```
is.numeric(inflation_rate)
```

Just like for the variable `num_shares`, this will return `TRUE` if the variable `inflation_rate` is of type numeric.

- For the variable `interest_rate_rising`:

```
is.logical(interest_rate_rising)
```

This will return `TRUE` if the variable `interest_rate_rising` is of type logical.

These examples illustrate how you can use the `is.*` functions to check the type of variables, helping ensure that you are handling data correctly in your analyses.

Another important family of functions is the `as.*` functions. They are used to convert an object from one type to another. They allow changing the data type of a variable, which can be useful in various situations, such as performing specific operations that require a certain data type or ensuring data type consistency in the code.

Some of the most common `as.*` functions include:

- `as.character()` converts to character:

```
number <- 123  
number_character <- as.character(number)
```

- `as.numeric()` converts to numeric:

```
text <- "3.14"  
number <- as.numeric(text)
```

- `as.logical()` converts to logical:

```
number <- 0  
logical <- as.logical(number)
```

These functions are useful for ensuring that data types are correct in your code and for guaranteeing that you can perform the desired operations on your objects.

However, it is important to note that not all conversions may be successful, especially when there is a loss of information (e.g., when converting from character to numeric).

Therefore, it is always a good practice to verify if the conversion was done correctly and if the resulting data is as expected.

Here is an example of converting a character to numeric with non-numeric text:

```
text <- "abc"  
number <- as.numeric(text)
```

Warning: NAs introduzidos por coerção

In this case, the conversion will fail, returning NA (Not Available) and issuing a *warning* message.

1.6 Data structures

In any data analysis, it is common to work with datasets that have different structures and formats.

Let's explore four fundamental data structures in R: vector, matrix, list, and DataFrame.

1.6.1 Vectors

A vector in R is a one-dimensional data structure with elements of the same type. Use `c()` to create vectors:

```
# Stock prices vector  
stock_prices <- c(100, 110, 105, 120, 115)
```

In some cases, it is useful to define sequences of numbers using the `:` operator and the `seq()` function.

```
# Sequence from 1 to 10:  
sequence <- 1:10  
sequence
```

```
[1] 1 2 3 4 5 6 7 8 9 10
```

```
# Sequence from 1 to 10 with an increment of 2:  
increment_sequence <- seq(1, 10, by = 2)  
increment_sequence
```

```
[1] 1 3 5 7 9
```

To check the size of a vector, you can use the `length()` function.

```
# checking the size of the stock prices vector  
length(stock_prices)
```

```
[1] 5
```

```
length(1:10)
```

```
[1] 10
```

To access elements in a vector in R, you can use numeric or logical indices inside square brackets `[]`.

You can access elements using numeric indices within square brackets `[]`. For example, `vector[i]` accesses the element at position `i` of `vector`.

```
# Stock prices vector  
stock_prices <- c(100, 110, 105, 120, 115)  
  
# Accessing the second element of the vector  
second_element <- stock_prices[2]
```

```
# Accessing a range of elements from the vector
multiple_elements <- stock_prices[3:5]
```

You can also access elements using logical indices inside square brackets [].

For example, `vector[logical_indices]` returns the elements of the vector where the logical indices are `TRUE`.

```
# Accessing stock prices greater than 110
prices_above_110 <- stock_prices[stock_prices > 110]
```

1.6.2 Matrices

A matrix in R is a two-dimensional data structure consisting of rows and columns of elements of the same type.

It is useful for representing tabular datasets, such as time series data or covariance matrices.

```
# Asset returns matrix
asset_returns <- matrix(c(0.05, 0.03, 0.02, 0.04, 0.06, 0.03),
                        nrow = 2, byrow = TRUE)
rownames(asset_returns) <- c("Stock 1", "Stock 2")
colnames(asset_returns) <- c("Year 1", "Year 2", "Year 3")
```

The code above creates a matrix called `asset_returns` that stores the returns of two assets over three years.

The `matrix()` function is used to create the matrix.

The vector `c(0.05, 0.03, 0.02, 0.04, 0.06, 0.03)` contains the return values, filled row by row.

Parameters `nrow = 2` and `byrow = TRUE` specify that the matrix should have 2 rows (to represent two assets), and values are filled by row.

The `rownames()` and `colnames()` functions assign names to the rows (“Stock 1” and “Stock 2”) and columns (“Year 1”, “Year 2”, “Year 3”).

The `class()` function returns the class of the object, which in this case will be “matrix,” indicating that `asset_returns` is a matrix in R.

The `dim()` function returns the dimensions of the matrix, i.e., the number of rows and columns.

```
# Verifying the matrix dimensions
dim(asset_returns)
```

```
[1] 2 3
```

This will output [2, 3], indicating that the matrix has 2 rows and 3 columns.

The `length()` function returns the total number of elements in an object.

For a matrix, this will return the total number of elements, which is the product of the number of rows and columns.

```
length(asset_returns)
```

```
[1] 6
```

To access rows, columns, and elements in a matrix in R, you can use numeric indices or names (if defined).

Here's how to do it:

- **Accessing Rows and Columns:** You can access rows and columns using numeric indices inside square brackets `[]`. For example, `matrix[i,]` accesses row `i`, and `matrix[, j]` accesses column `j`. To access a specific cell, you use `matrix[i, j]`, where `i` is the row number and `j` is the column number.

```
# Accessing the first row of the matrix
first_row <- asset_returns[1, ]

# Accessing the second column of the matrix
second_column <- asset_returns[, 2]

# Accessing the element in the second row and third column
element <- asset_returns[2, 3]

# Accessing more than one column
selected_columns <- asset_returns[, c(1, 3)]
```

- **Accessing Rows and Columns by name:** If row or column names are defined, you can also use them to access data:


```
# Accessing the row named "Stock 1"
stock1 <- asset_returns["Stock 1", ]

# Accessing the column named "Year 2"
year2 <- asset_returns[, "Year 2"]

# Accessing the element in "Stock 2" and "Year 3"
specific_element <- asset_returns["Stock 2", "Year 3"]
```

In R, unlike other programming languages, the indices of rows and columns in matrices (as well as in vectors, lists, etc.) start at 1 instead of 0.

This means the first element of a matrix is at index 1, the second at index 2, and so on.

1.6.3 Lists

In R, a list is a flexible data structure that can store elements of different types, such as vectors, matrices, other lists, or even functions. Lists are useful when you need to store and manipulate heterogeneous data sets or complex structures.

We can create a list that stores information about a country, such as its name, GDP, inflation rate, and a time series of exchange rate values.

```
# Creating a list with country information
country_info <- list(
  name = "Brazil",
  gdp = 1609,
  inflation = 0.05,
  exchange_rates = c(4.86, 5.13, 5.20, 5.07, 4.97)
)
```

In this example, `country_info` is a list containing four elements:

- **name:** the name of the country (character type).
- **gdp:** the country's Gross Domestic Product (numeric type).
- **inflation:** the country's inflation rate (numeric type).
- **exchange_rates:** a time series of the country's exchange rate values (numeric vector type).

This list exemplifies how we can store different types of data in a list in R. It can be used to represent a country's economic information in an organized and accessible manner.

To access individual elements in a list by name, we use the dollar operator `$`.

```
# Accessing the name of the country
country_info$name
```

```
[1] "Brazil"
```

```
# Accessing the GDP
country_info$gdp
```

```
[1] 1609
```

```
# Accessing exchange rates
country_info[["exchange_rates"]][3]
```

```
[1] 5.2
```

We can also access individual elements in a list by index using square brackets `[]`.

```
# Accessing the first element of the list (country name)
first_element <- country_info[[1]]

# Accessing the third element of the list (inflation rate)
third_element <- country_info[[3]]
```

You may have noticed the use of double brackets to access list elements. In R, single brackets (`[]`) and double brackets (`[[]]`) serve different purposes when accessing elements in a list.

Understanding Lists Through the Train Analogy

Imagine a list in R as a train, and each element of the list is a wagon of the train. Now, inside each wagon, you can store different types of cargo, such as boxes, sacks, or even other wagons.

For instance, in one wagon, you might have a vector; in another, a matrix; and in another, just a single number.

Each element of the list can be different in type and content, just as each wagon of a train can hold different things.

Accessing a Specific Wagon

To access specific information about Brazil, such as the country name or GDP, we use single brackets [].

For example:

- `country_info["nome"]`: we get the wagon containing the country name, which is “Brazil”.
- `country_info[2]`: we get the second wagon, which contains the GDP, with a value of 1609.

Accessing Elements Inside a Wagon

If we want to access specific elements inside a wagon, we use double brackets [[]].

For example:

- `country_info[["exchange_rates"]]`: we open the wagon containing the country’s exchange rate information and access its contents, which is a vector of different exchange rate values over time.

To access a specific value from this vector, we can use single brackets [] again:

- `country_info[["exchange_rates"]][3]`: this opens the wagon containing the exchange rate vector and retrieves the third value, which is 5.20.

Using the `class()` function, we can observe the difference between objects obtained with single and double brackets.

```
class(country_info["nome"])
```

```
[1] "list"
```

```
class(country_info[["nome"]])
```

```
[1] "NULL"
```

1.6.4 :::

In summary, single brackets are used to access subsets of elements in a list while preserving their structure, whereas double brackets are used to access individual values from a list, without preserving the original structure.

:::

1.6.5 DataFrames

DataFrames are tabular structures in R where columns can have different types.

You can create one using the `data.frame()` function:

```
# Creating a dataframe with economic data
economic_data <- data.frame(
  country = c("Brazil", "USA", "China", "India", "Japan"),
  continent = factor(c("America", "America", "Asia", "Asia", "Asia")),
  population = c(213, 328, 1441, 1380, 126),
  gdp_per_capita = c(10294, 65741, 10380, 2353, 41581),
  inflation = c(0.02, 0.01, 0.04, 0.06, 0.005)
)
```

We can access individual elements, rows, or columns of a dataframe using numeric indices or column names.

```
# Accessing the first row of the dataframe
(first_row <- economic_data[1, ])
```

```
country continent population gdp_per_capita inflation
1 Brazil America 213 10294 0.02
```

```
# Accessing the "country" column of the dataframe
(country <- economic_data$country)
```

```
[1] "Brazil" "USA" "China" "India" "Japan"
```

```
# Accessing the element in the second row and third column of the dataframe
(element <- economic_data[2, 3])
```

```
[1] 328
```

We can combine dataframes based on common columns using the `merge()` function.

```
# Creating another dataframe for merging
demographic_data <- data.frame(
  country = c("China", "India", "Japan", "Brazil", "USA"),
  life_expectancy = c(76, 69, 84, 75, 79)
)

# Performing a merge based on the "country" column
(data_and_demographic_info <- merge(economic_data, demographic_data, by = "country"))
```

	country	continent	population	gdp_per_capita	inflation	life_expectancy
1	Brazil	America	213	10294	0.020	75
2	China	Asia	1441	10380	0.040	76
3	India	Asia	1380	2353	0.060	69
4	Japan	Asia	126	41581	0.005	84
5	USA	America	328	65741	0.010	79

We can also add new rows of data to an existing dataframe.

```
# Creating another dataframe to combine rows
more_data <- data.frame(
  country = c("South Africa", "Germany"),
  continent = c("Africa", "Europe"),
  population = c(60, 83),
  gdp_per_capita = c(6151, 52947),
  inflation = c(0.025, NA),
  life_expectancy = c(58, 81)
)

# Combining dataframes by rows
(all_data <- rbind(data_and_demographic_info, more_data))
```

	country	continent	population	gdp_per_capita	inflation	life_expectancy
1	Brazil	America	213	10294	0.020	75
2	China	Asia	1441	10380	0.040	76
3	India	Asia	1380	2353	0.060	69
4	Japan	Asia	126	41581	0.005	84
5	USA	America	328	65741	0.010	79
6	South Africa	Africa	60	6151	0.025	58
7	Germany	Europe	83	52947	NA	81

! Missing Data

You may have noticed in the `more_data` dataframe that the inflation value for Germany is marked as `NA`.

An `NA` value, short for “Not Available,” indicates a missing value in a dataset.

In the example above, the presence of `NA` in the inflation column for Germany means that no inflation value is available for this country in the provided table.

A value can be marked as `NA` in various situations, including:

- **Missing data:** When no information is available for a specific field in a dataset.
Example: Lack of data on unemployment rates in certain regions due to unavailability or incomplete reporting.
- **Measurement or collection errors:** Errors during the measurement or data collection process can result in inaccurate or missing values.
Example: Recording a country’s GDP might lead to missing values for certain quarters due to collection errors.
- **Inapplicable values:** Some variables might not apply to all cases.
Example: In analyzing government spending on education, some countries might not have data due to differences in reporting policies or lack of investment in education.
- **Unrecorded values:** In some databases, specific values might not be recorded intentionally, either for privacy reasons or because they are irrelevant to the analysis.
Example: Collecting data on individuals’ net worth in an income survey might result in some participants opting not to disclose financial information for privacy reasons. In such cases, the corresponding values would be marked as `NA`.

1.7 Exercises

1. **Consider the following economic sectors:** “Finance,” “Information Technology,” “Industrial Goods,” and “Healthcare.” Generate a random sample of size 1,000 from these sectors with equal probability for each. Display the first few values of the resulting variable and count how many companies belong to each economic sector.

2 Execution Flows

2.1 Conditional Structures

Code flow in R can be controlled using conditional structures, such as `if`, `else if`, and `else`. These structures allow you to execute different code blocks based on specific conditions.

2.1.1 `if` and `else`

The `if` structure is a control flow structure that executes a block of code if a specified condition is true. If the condition is false, the code block within the `if` will not be executed. On the other hand, `else` is used to execute a block of code when the `if` condition is false.

The basic syntax of `if` and `else` in R is as follows:

```
if (condition) {  
  # Code block to execute if the condition is true  
} else {  
  # Code block to execute if the condition is false  
}
```

Here is a practical example of how to use `if` and `else` to check if a randomly chosen integer between -10 and 10 is positive or negative:

```
# Setting the seed to ensure reproducibility  
set.seed(42)  
  
# Generating a random number between -10 and 10  
number <- sample(-10:10, 1)  
  
if (number > 0) {  
  print("The number is positive.")  
} else {  
  print("The number is negative or zero.")  
}
```

```
[1] "The number is positive."
```

In this example, `sample(-10:10, 1)` generates a random number between -10 and 10, and the value is assigned to the `number` variable. Additionally, `set.seed(42)` sets the seed to 42. This ensures that when generating the random number with `sample()`, the same number is chosen every time the code is executed. Then, we check if the number is positive or not and print the corresponding message.

2.1.2 else if

In addition to `if` and `else`, we can also use `else if` to add more conditions to the conditional structure. The `else if` allows checking multiple conditions in sequence. If the `if` condition is false, it checks the next `else if` condition. If all the `if` and `else if` conditions are false, the code block within the `else` is executed.

Here is the syntax of `else if`:

```
if (condition1) {  
  # Code block to execute if condition1 is true  
} else if (condition2) {  
  # Code block to execute if condition2 is true  
} else {  
  # Code block to execute if none of the previous conditions are true  
}
```

Here is a practical example of how to use `if`, `else if`, and `else` to evaluate a company's performance based on its annual revenue:

```
# Determining the company's classification based on annual revenue  
annual_revenue <- 1500000  
  
if (annual_revenue >= 2000000) {  
  print("Large Company")  
} else if (annual_revenue >= 1000000) {  
  print("Medium Company")  
} else if (annual_revenue >= 500000) {  
  print("Small Company")  
} else {  
  print("Microenterprise")  
}
```

```
[1] "Medium Company"
```


In this example, the company is classified based on its annual revenue. If the revenue is equal to or greater than 2,000,000, the company will be classified as a “Large Company”. If it is between 1,000,000 and 1,999,999, it will be classified as a “Medium Company”. If it is between 500,000 and 999,999, it will be classified as a “Small Company”. Otherwise, it will be considered a “Microenterprise”.

2.2 Loop Structures

Loop structures, also known as repetition structures, are used to execute a block of code repeatedly while a specific condition is true or to iterate over a sequence of elements. This is useful when you need to perform a task multiple times or want to loop through a collection of data.

2.2.1 for

One of the most common repetition structures is the `for` loop. The `for` loop is used to iterate over a sequence of values, such as a numeric sequence of integers or the elements of a vector.

There are two ways to use the `for` loop.

- Using `for` to iterate over indices:

```
# Example of a for loop to iterate over indices
for (i in 1:5) {
  print(i)
}
```

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

In this example, the `for` loop iterates over the values from 1 to 5. In the first iteration, `i` is equal to 1; in the second iteration, `i` is equal to 2; and so on, until `i` is equal to 5.

- Using `for` to iterate over elements:

```
# Example of a for loop to iterate over elements of a vector
customers <- c("John", "Mary", "Joseph", "Anna")
for (name in customers) {
```

```
    print(name)
  }
```

```
[1] "John"
[1] "Mary"
[1] "Joseph"
[1] "Anna"
```

In the example above, the `for` loop iterates over the elements of the `customers` vector. In the first iteration, `name` is equal to “John”; in the second iteration, `name` is equal to “Mary”; and so on, until all elements of the vector are processed.

In both examples, the code block inside the `for` loop is executed repeatedly for each value of `i` (in the first example) or `name` (in the second example) until the sequence is completely traversed.

In the example below, let’s simulate economic data for 10 fictitious countries and calculate the per capita GDP of each country.

```
set.seed(42)
gdp_countries <- runif(10, min = 25000000, max = 40000000)
population_countries <- runif(10, min = 1000000, max = 15000000)

gdp_per_capita <- numeric(length = 10)

# For loop to calculate per capita GDP for each country
for (i in 1:10) {
  # Calculating per capita GDP
  gdp_per_capita[i] <- gdp_countries[i] / population_countries[i]
}
print(round(gdp_per_capita, 3))
```

```
[1] 5.227 3.529 2.080 8.185 4.634 2.315 2.453 10.216 4.556 4.022
```

💡 Click to see an extra example

Imagine we have a time series representing the daily closing price of a stock over a period of 30 days. We want to calculate the 5-day moving average of this price, that is, for each day, we want to calculate the average of the closing prices of the five previous days, including the current day.

First, let’s simulate the data for the daily closing price of the stock:

```
set.seed(42)
stock_price <- runif(30, min = 9, max = 15)
```

Now, let's calculate the 5-day moving average using a for loop:

```
moving_average <- numeric(length = 26) # Vector to store the moving average

for (i in 5:30) {
  moving_average[i - 4] <- mean(stock_price[(i - 4):i])
}
```

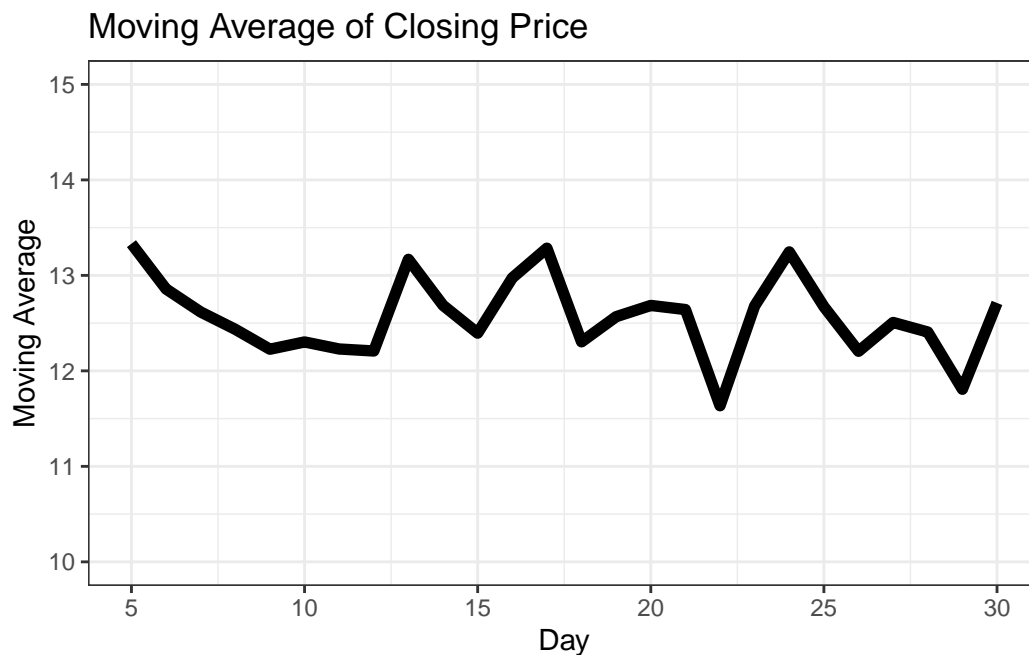
In this for loop, we start from the fifth day, as we need at least five days to calculate the 5-day moving average. For each day from the fifth to the thirtieth day, we calculate the average of the closing prices of the five previous days, including the current day, and store this value in the `moving_average` vector.

Now we can print the calculated moving average:

```
print(moving_average)
```

```
[1] 13.33226 12.85740 12.61682 12.43505 12.22691 12.30289 12.22926 12.20829
[9] 13.16830 12.68642 12.39510 12.97382 13.28476 12.30414 12.56762 12.68527
[17] 12.64209 11.63467 12.68036 13.24636 12.67289 12.20510 12.50690 12.40711
[25] 11.80747 12.71175
```

This example demonstrates how to use a for loop in conjunction with vectors to calculate the moving average of a time series. The chart below shows the moving average over the days.



i Note

You will learn how to build charts like this in [Chapter 4](#).

2.2.2 while

The **while** structure is used to repeat a block of code as long as a specified condition is true. Here is the general structure of a while loop:

```
while (condition) {  
    # Code to be repeated while the condition is true  
}
```

The condition is a logical expression that is evaluated before each execution of the code block inside the loop. If the condition is true, the code block is executed; if false, the loop terminates, and control passes to the next line of code after the loop.

In the example below, we define a vector called **actions**, which contains a list of possible activities a person might perform during the day. One of the possible actions will be chosen randomly.

```

actions <- c("Learn to code in R",
             "Learn to code in Python",
             "Make coffee",
             "Rest")

set.seed(42)
action <- sample(actions, 1)
print(action)

```

```
[1] "Learn to code in R"
```

In the following code snippet, we use the `while` structure to continue randomly selecting an activity from the `actions` vector until the selected activity is “Rest”. The loop starts by checking if the variable `action` is different from “Rest”. If this condition is true, a new activity is randomly selected from the `actions` vector using the `sample()` function with `size = 1`. The selected activity is then printed to the screen using the `print()` function. This process repeats until the selected activity is “Rest”, at which point the loop ends.

```


set.seed(420)
while (action != "Rest") {
  action <- sample(actions, 1)
  print(action)
}

```

```

[1] "Learn to code in R"
[1] "Learn to code in R"
[1] "Learn to code in Python"
[1] "Learn to code in Python"
[1] "Rest"

```

 Click to see an additional example

Let's consider an example where we want to simulate population growth over time, where the number of periods required to reach a certain limit is unknown. In this case, we use a `while` loop to simulate population growth until the population reaches a set threshold.

```

set.seed(42) # Set seed for reproducibility

# Initial population
population <- 1000

# Annual population growth rate (in decimal)
growth_rate <- 0.02

# Desired population limit
population_limit <- 2000

# Initialize year counter
years <- 0

# Simulating population growth until the limit is reached
while (population < population_limit) {
  # Calculate new individuals this year
  new_individuals <- population * growth_rate

  # Increment population with new individuals
  population <- population + new_individuals

  # Increment year counter
  years <- years + 1
}

# Print the number of years required to reach the population limit
print(paste("It took", years, "years to reach a population of", population))

```

```
[1] "It took 36 years to reach a population of 2039.8873437157"
```

i Note

You can calculate the exact number of years needed to reach the population limit directly. Using algebra, the formula is as follows:

$$\text{years} = \frac{\log\left(\frac{\text{population_limit}}{\text{initial_population}}\right)}{\log(1 + \text{growth_rate})}.$$

2.3 Functions

A function in R is a block of code that performs a specific task and can be reused multiple times. The syntax for defining a function in R follows this pattern:

```
function_name <- function(parameters) {  
  # Function body  
  # Code that performs the desired task  
  # May include mathematical operations, data manipulation, etc.  
  return(result) # Returns the desired result  
}
```

Parameters are variables that a function takes as input to perform its operations. They are specified within parentheses when defining the function. Inside the function body, these parameters can be used to perform calculations or operations.

The example below defines a function to perform a simple linear regression. The `linear_regression` function takes two parameters: `x` and `y`, which represent the input data for the regression. Inside the function, a linear regression model is created using the `lm()` function in R, with `y` as a function of `x`. The model is then returned as the function result.

```
# Function to perform simple linear regression  
linear_regression <- function(x, y) {  
  model <- lm(y ~ x) # Creating the linear regression model  
  return(model) # Returning the model  
}  
  
# Example data: salary (y) as a function of years of education (x)  
education_years <- c(10, 12, 14, 16, 18)  
salary <- c(2500, 3300, 3550, 3700, 4500)  
  
# Calling the linear regression function  
regression_model <- linear_regression(education_years, salary)
```

Display the summary of the trained model:

```
# Display regression results  
summary(regression_model)
```

Call:

```
lm(formula = y ~ x)
```

Residuals:

1	2	3	4	5
-130	230	40	-250	110

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	430.00	498.20	0.863	0.45156
x	220.00	34.88	6.307	0.00805 **

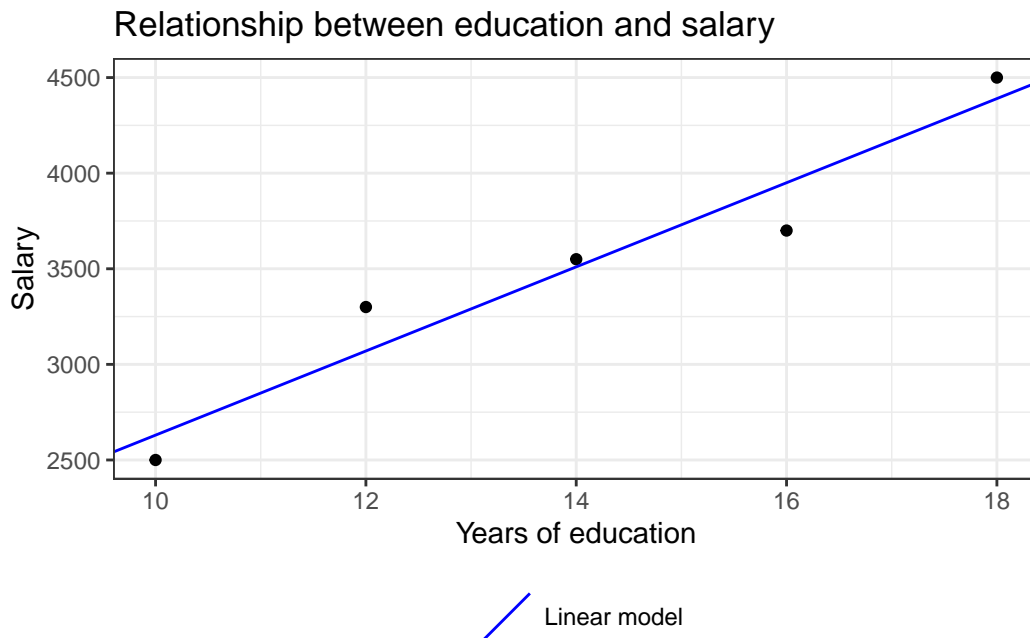
Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 220.6 on 3 degrees of freedom

Multiple R-squared: 0.9299, Adjusted R-squared: 0.9065

F-statistic: 39.78 on 1 and 3 DF, p-value: 0.008054

Below is a scatter plot showing the relationship between years of education and salary. The blue line represents the linear regression model trained on the data.



Note

You will learn how to build charts like this in [Chapter 4](#).

2.4 Packages

Packages in R are collections of functions, datasets, and documentation that extend the core capabilities of the R language. They are essential for performing a wide variety of tasks.

R packages are available on CRAN (Comprehensive R Archive Network), a centralized repository that hosts a vast collection of packages. To access CRAN packages, use the `install.packages()` function. For example:

```
install.packages("package_name")
```

After installation, load the package into your R session using the `library()` function:

```
library(package_name)
```

This makes the package's functions and datasets available for use in your current session.

2.5 Exercises

1. In this exercise, simulate a coin toss and store the results as a factor with levels “heads” and “tails”. Follow these steps:

a) Use the command below to generate random samples following a binomial distribution to simulate 100 coin tosses:

```
set.seed(42)
tosses <- rbinom(100, 1, 0.5)
```

b) Assume 0 represents “heads” and 1 represents “tails”. Create a variable to store the tosses as a factor with levels “heads” and “tails”.

c) Count how many times each result occurred in this experiment.

d) Use a loop to iterate backward through the vector and find the last “heads” result.

2. Working with package datasets.

a) Install the `nycflights13` package:

```
install.packages("nycflights13")
```

b) Load the package into your environment:

```
library(nycflights13)
```

c) Use the commands below to explore the contents of the `flights` and `airports` data frames:

```
?flights  
?airports
```

d) Filter the flights that occurred on 01/25/2013 and store them in the variable `christmas`.

e) How many flights departed from New York on 12/25/2013?

f) Get a summary of the `dep_delay` column. Are there missing values? If yes, remove them.

g) Find the name of the destination airport of the flight with the longest departure delay on 12/25/2013. Hint: Merge the `flights` and `airports` datasets. ““

3 Data Manipulation

Mastering data manipulation and processing techniques in R is essential for anyone working in data analysis or data science. The ability to clean, transform, and prepare data efficiently is crucial to ensure that analysis results are accurate and reliable. Moreover, mastering these techniques saves time and increases productivity.

3.1 Importing External Files

Two of the most common formats for storing not-so-large datasets are `csv` and `xlsx`.

When loading data from CSV files in R, two commonly used options are the `read.csv()` and `read_csv()` functions. Both are effective for importing tabular data but have significant differences. The `read.csv()` function is a standard option in base R, being simple to use and widely known. On the other hand, `read_csv()` is part of the `readr` package, offering optimized performance and automatic data type detection. While `read.csv()` tends to be slower, especially with large datasets, `read_csv()` is faster and more accurate, capable of maintaining column names as symbols and properly handling data, including empty strings.

```
# Using read.csv()
data_read_csv <- read.csv("data.csv")

# Using read_csv() from the readr package
library(readr)
data_readr <- read_csv("data.csv")
```

To import data from an Excel file (`xlsx` format) in R, we can use the `readxl` library. First, you need to install it using the command `install.packages("readxl")`. Then, you can use the `read_excel()` function to read the data. For example:

```
library(readxl)
data <- read_excel("file.xlsx")
```

💡 Setting Your Working Directory

It is good practice to set a working directory in your R scripts because it helps keep things organized and makes it easier to access data files and results. By setting a working directory, you ensure that all files referenced in your scripts will be easily found without needing to specify long absolute paths.

To set the working directory in R, you can use the `setwd()` function. For example, if you want to set the directory to “C:/MyDirectory”, you can do the following:

```
setwd("C:/MyDirectory")
```

You can also set the directory using the RStudio interface. Simply select “Session” from the menu, then “Set Working Directory,” and finally “Choose Directory.” This will open a dialog box where you can navigate to the desired directory and select it. After selecting the directory, it will become the current working directory.

3.2 The tidyverse Package

The `tidyverse` package is a collection of R packages designed to work seamlessly and intuitively for data analysis. It includes a variety of powerful and popular packages such as `ggplot2`, `dplyr`, `tidyr`, `tibble`, `readr`, `purrr`, `forcats`, and `stringr`. Each package in the `tidyverse` is designed to handle a specific step of the data analysis workflow, from importing and cleaning to visualization and modeling. All packages in the `tidyverse` share a common underlying design philosophy, grammar, and data structures. Learn more [on the package page](#).

3.3 The |> Pipe Operator

The `|>` operator, known as the pipe, is a powerful tool in R that facilitates chaining operations in sequence. It allows you to write code more clearly and concisely, especially when working with `tidyverse` packages. The pipe takes the result of an expression on the left and passes it as the first argument to the next expression on the right.

💡 Tip

You don’t need to type `|>` every time you need it. Use the shortcut `Ctrl+Shift+M`.

Suppose we have a function `f`, a function `g`, and a variable `x`. We want to apply `g` to `x`, and then apply `f` to the result. Here’s how we could do this in two ways: using the traditional chained approach and using the `|>` pipe.

```
result <- f(g(x))
```

```
x |>  
  g() |>  
  f()
```

Both methods will produce the same result. However, the second approach using the `|>` pipe is more readable and easier to understand, especially when chaining multiple operations. This makes the code more concise and closer to a natural reading of the operation being performed.

Tip

A good practice when using the `|>` pipe is to break the line after each pipe to improve code readability.

3.4 Data in *Tidy* Format

“Tidy datasets are all alike, but every messy dataset is messy in its own way.” —
Hadley Wickham.

The same dataset can be represented in various ways. See the code below that shows the same data in three different formats.

```
table1
```

```
# A tibble: 6 x 4  
  country    year cases population  
  <chr>      <dbl> <dbl>      <dbl>  
1 Afghanistan 1999    745   19987071  
2 Afghanistan 2000   2666   20595360  
3 Brazil      1999  37737   172006362  
4 Brazil      2000  80488   174504898  
5 China       1999 212258  1272915272  
6 China       2000 213766  1280428583
```

```
table2
```

```
# A tibble: 12 x 4
```

	country	year	type	count
	<chr>	<dbl>	<chr>	<dbl>
1	Afghanistan	1999	cases	745
2	Afghanistan	1999	population	19987071
3	Afghanistan	2000	cases	2666
4	Afghanistan	2000	population	20595360
5	Brazil	1999	cases	37737
6	Brazil	1999	population	172006362
7	Brazil	2000	cases	80488
8	Brazil	2000	population	174504898
9	China	1999	cases	212258
10	China	1999	population	1272915272
11	China	2000	cases	213766
12	China	2000	population	1280428583

```
table4a
```

```
# A tibble: 3 x 3
  country    `1999` `2000`
  <chr>      <dbl> <dbl>
1 Afghanistan    745   2666
2 Brazil        37737  80488
3 China        212258 213766
```

All of the above representations are of the same data, but they are not equally easy to use. `table1`, for example, will be much more accessible for working within the tidyverse due to its organization in the *tidy* format. There are three interrelated rules that characterize a dataset in *tidy* format:

1. Each variable is a column; each column represents a variable.
2. Each observation is a row; each row represents an observation.
3. Each value is a cell; each cell contains a single value.

The figure below graphically represents this concept.

Pivoting data is the process of reorganizing a dataset to make it compatible with the *tidy* format. This involves transforming the data from a wider format to a longer format, or vice versa, to ensure that each variable corresponds to a column and each observation to a row.

In the example below, we are transforming the data from `table2` to a wider format, where each unique value of the variable `type` becomes a new column. Note that each unit of information (country, year, cases, and count) is split into two rows. This operation makes the data table wider, ensuring that each unit of data is represented in a single row.

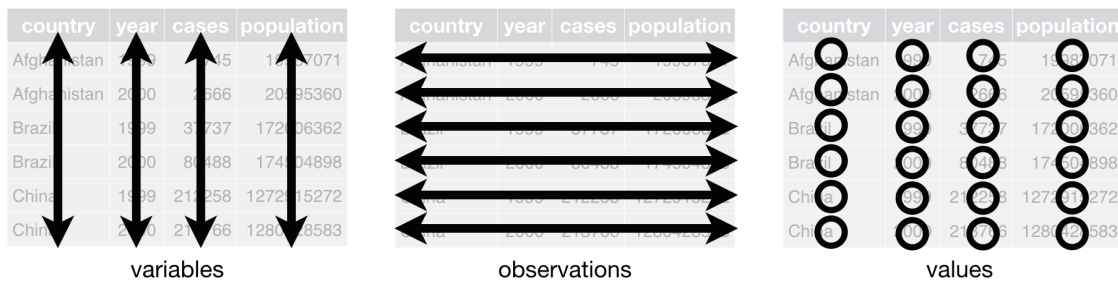


Figure 3.1: Image from the R4DS book.

```
table2 |>
  pivot_wider(names_from="type", values_from="count")
```

```
# A tibble: 6 x 4
  country    year  cases population
  <chr>      <dbl> <dbl>      <dbl>
1 Afghanistan 1999     745   19987071
2 Afghanistan 2000    2666   20595360
3 Brazil      1999   37737   17206362
4 Brazil      2000   80488   174504898
5 China       1999  212258  1272915272
6 China       2000  213766  1280428583
```

In the example below, we are transforming the data from `table4a` to a longer format, where the columns representing specific years (1999 and 2000) are gathered into a single column called `year`, and the corresponding values are placed in a new column called `cases`. In this case, the information about years was stored as column names, but according to the *tidy* data principle, they should be in columns. Therefore, we use the `pivot_longer` function.

```
table4a |>
  pivot_longer(cols = c(`1999`, `2000`), names_to = "year", values_to = "cases")
```

```
# A tibble: 6 x 3
  country    year  cases
  <chr>      <chr> <dbl>
1 Afghanistan 1999     745
2 Afghanistan 2000    2666
```

3	Brazil	1999	37737
4	Brazil	2000	80488
5	China	1999	212258
6	China	2000	213766

The two functions `pivot_wider` and `pivot_longer` are sufficient for transforming datasets into *tidy* format.

3.5 Main Verbs of the dplyr Package

The `dplyr` package is one of the most powerful tools for data manipulation in the R environment. It offers a cohesive set of functions that simplify common manipulation tasks such as filtering, selecting, grouping, sorting, and summarizing data. `dplyr` uses an intuitive and consistent syntax, making it easy to write clean and readable code.

Let's study the functionality of the main verbs of the package. To illustrate, we will use the `gapminder` dataset. It is a collection of socioeconomic information for various countries over time. See Rosling (2012). It includes variables such as life expectancy, GDP per capita, infant mortality rate, and population size for different countries and years, covering several decades.

To load the `gapminder` dataset, you need to load the `gapminder` package. With the `gapminder` package loaded, the `gapminder` dataset will be available for use in your R environment:

```
library(gapminder)
```

Warning: package 'gapminder' was built under R version 4.2.3

```
head(gapminder)
```

The `glimpse()` function provides a quick and concise overview of the structure of a dataset. When applied to a dataset, like `gapminder`, it displays essential information about the variables present, including the number of rows, columns, and the first few rows of the dataset:

```
glimpse(gapminder)
```

```
Rows: 1,704
```

```
Columns: 6
```

```
$ country   <fct> "Afghanistan", "Afghanistan", "Afghanistan", "Afghanistan", ~
$ continent <fct> Asia, Asia, Asia, Asia, Asia, Asia, Asia, Asia, Asia, ~
```



```
$ year      <int> 1952, 1957, 1962, 1967, 1972, 1977, 1982, 1987, 1992, 1997, ~
$ lifeExp   <dbl> 28.801, 30.332, 31.997, 34.020, 36.088, 38.438, 39.854, 40.8~
$ pop       <int> 8425333, 9240934, 10267083, 11537966, 13079460, 14880372, 12~
$ gdpPercap <dbl> 779.4453, 820.8530, 853.1007, 836.1971, 739.9811, 786.1134, ~
```

3.5.1 select

The `select()` verb is used to select specific columns from a dataset. With `select()`, you can choose the desired columns based on their names, data types, or other criteria.

For example, considering the `gapminder` dataset, suppose we want to select only the columns referring to the year, country, life expectancy, and GDP per capita. We can do this as follows:

```
# Selecting columns by name
gapminder |>
  select(year, country, lifeExp, gdpPercap)

# Selecting only numeric columns
gapminder |>
  select(where(is.numeric))

# Selecting columns that start with "co"
gapminder |>
  select(starts_with("co"))
```

Note that in the above examples, none of the selections was saved to a variable. To save the selections to a variable, you can assign the result of each `select()` operation to a separate variable. For example:

```
gapminder_character <- gapminder |>
  select(where(is.character))
```

3.5.2 arrange

The verb `arrange()` is used to reorder the rows of a dataset based on the values of one or more columns. When applied to a dataset, `arrange()` sorts the rows in ascending or descending order based on the specified column values.

In the first example using the verb `select()`, we can sort the data by country in alphabetical order as follows:

```
gapminder |>
  select(year, country, lifeExp, gdpPercap, pop) |>
  arrange(country)
```

```
# A tibble: 1,704 x 5
   year country    lifeExp gdpPercap      pop
  <int> <fct>        <dbl>    <dbl>    <int>
1  1952 Afghanistan  28.8      779.  8425333
2  1957 Afghanistan  30.3      821.  9240934
3  1962 Afghanistan  32.0      853. 10267083
4  1967 Afghanistan  34.0      836. 11537966
5  1972 Afghanistan  36.1      740. 13079460
6  1977 Afghanistan  38.4      786. 14880372
7  1982 Afghanistan  39.9      978. 12881816
8  1987 Afghanistan  40.8      852. 13867957
9  1992 Afghanistan  41.7      649. 16317921
10 1997 Afghanistan  41.8      635. 22227415
# i 1,694 more rows
```

In the example below, we are organizing the data by year in ascending order and life expectancy in descending order within each year.

```
gapminder |>
  select(year, country, lifeExp, gdpPercap, pop) |>
  arrange(year, desc(lifeExp))
```

```
# A tibble: 1,704 x 5
   year country    lifeExp gdpPercap      pop
  <int> <fct>        <dbl>    <dbl>    <int>
1  1952 Norway      72.7    10095.  3327728
2  1952 Iceland     72.5     7268.   147962
3  1952 Netherlands  72.1     8942. 10381988
4  1952 Sweden      71.9     8528.  7124673
5  1952 Denmark     70.8     9692.  4334000
6  1952 Switzerland  69.6    14734.  4815000
7  1952 New Zealand  69.4    10557.  1994794
8  1952 United Kingdom 69.2     9980. 50430000
9  1952 Australia    69.1    10040.  8691212
10 1952 Canada      68.8    11367. 14785584
# i 1,694 more rows
```

💡 Tip

When using the verb `select()` with the prefix `-`, you can specify the columns you want to **exclude** from the dataset. In the example below, we exclude the `continent` column from the selection.

```
gapminder |>
  select(-continent)

# A tibble: 1,704 x 5
  country      year lifeExp      pop gdpPercap
  <fct>      <int>   <dbl>   <int>   <dbl>
1 Afghanistan 1952    28.8  8425333    779.
2 Afghanistan 1957    30.3  9240934    821.
3 Afghanistan 1962    32.0 10267083    853.
4 Afghanistan 1967    34.0 11537966    836.
5 Afghanistan 1972    36.1 13079460    740.
6 Afghanistan 1977    38.4 14880372    786.
7 Afghanistan 1982    39.9 12881816    978.
8 Afghanistan 1987    40.8 13867957    852.
9 Afghanistan 1992    41.7 16317921    649.
10 Afghanistan 1997    41.8 22227415    635.
# i 1,694 more rows
```

3.5.3 filter

To analyze specific data of interest, it is often necessary to filter the dataset to include only relevant observations. The verb `filter()` is used for this purpose. Simply define one or more logical conditions that the rows of the dataset must satisfy to be displayed.

In the example below, we filter the data to include only observations where the country is either Brazil or Argentina.

```
gapminder |>
  select(year, country, lifeExp, gdpPercap, pop) |>
  arrange(year, desc(lifeExp)) |>
  filter(country == "Brazil" | country == "Argentina")
```

```
# A tibble: 24 x 5
  year country      lifeExp gdpPercap      pop
  <int> <fct>      <dbl>   <dbl>   <int>
1 1997 Brazil    75.9    12240  206137368
2 1997 Argentina 75.1    10180  37413736
```

```

1 1952 Argentina    62.5    5911.  17876956
2 1952 Brazil       50.9    2109.  56602560
3 1957 Argentina    64.4    6857.  19610538
4 1957 Brazil       53.3    2487.  65551171
5 1962 Argentina    65.1    7133.  21283783
6 1962 Brazil       55.7    3337.  76039390
7 1967 Argentina    65.6    8053.  22934225
8 1967 Brazil       57.6    3430.  88049823
9 1972 Argentina    67.1    9443.  24779799
10 1972 Brazil      59.5    4986. 100840058
# i 14 more rows

```

3.5.4 mutate

The verb `mutate()` is used to create or modify columns in an existing dataset. It allows adding new variables calculated based on existing variables or modifying existing variables according to specific logic.

For example, we can use `mutate()` to calculate a new variable representing the total GDP of each country by multiplying per capita GDP by population size. Here's an example of how to do this with the `gapminder` dataset:

```

gapminder_total_gdp <- gapminder |>
  select(country, year, lifeExp, gdpPercap, pop) |>
  mutate(total_gdp = gdpPercap * pop)

```

3.5.5 summarise

The verb `summarise()` is used to summarize data into a single row, usually by calculating summary statistics such as mean, sum, median, etc. It allows calculating statistical summaries in a dataset, creating a new table containing the summarized results.

Here is an example of how to use `summarise()` to calculate the average life expectancy using the `gapminder` dataset:

```

gapminder |>
  summarise(mean_lifeExp = mean(lifeExp, na.rm = TRUE))

# A tibble: 1 x 1
  mean_lifeExp
    <dbl>

```

1 59.5

3.5.6 group by

The verb `group_by()` is used to split data into groups based on the values of one or more variables. It does not perform calculations by itself but changes the behavior of summary functions, such as `summarise()`, to operate separately on each group.

Here is an example of how to use `group_by()` with the `gapminder` data to calculate the average life expectancy by continent:

```
gapminder |>
  group_by(continent) |>
  summarise(mean_lifeExp = mean(lifeExp, na.rm = TRUE))
```

```
# A tibble: 5 x 2
  continent mean_lifeExp
  <fct>      <dbl>
1 Africa      48.9
2 Americas    64.7
3 Asia        60.1
4 Europe      71.9
5 Oceania     74.3
```

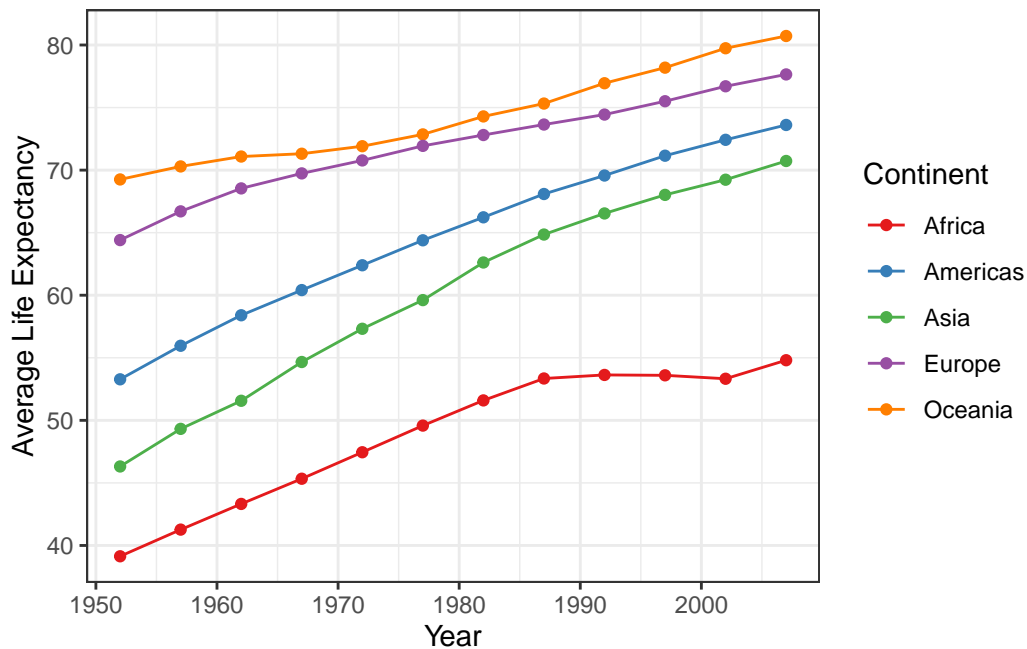
The example below uses all the main verbs of `dplyr` to calculate the average life expectancy and the average GDP (in thousands) by continent in the year 2007.

```
gapminder |>
  select(country, continent, year, lifeExp, gdpPercap) |>
  filter(year == 2007) |> # data only for the year 2007
  mutate(gdp = gdpPercap / 1000) |> # represents per capita GDP in thousands
  group_by(continent) |> # group the data by continent
  summarise(mean_lifeExp = mean(lifeExp, na.rm = TRUE), # average life expectancy
            mean_gdp = mean(gdp, na.rm = TRUE)) |> # average per capita GDP in billions
  arrange(desc(mean_lifeExp))
```

```
# A tibble: 5 x 3
  continent mean_lifeExp mean_gdp
  <fct>      <dbl>      <dbl>
1 Oceania      80.7      29.8
2 Europe       77.6      25.1
```

3 Americas	73.6	11.0
4 Asia	70.7	12.5
5 Africa	54.8	3.09

The graph below shows the evolution of average life expectancy in the continents over the years.



💡 Challenge

What change was made to the code of the previous example to construct the data used in this graph?

3.6 Helper Functions

Introducing auxiliary functions from the `dplyr` package that can be very useful in various contexts.

- `pull`, `distinct`, `unite`, `separate_wider_delim`, and the family of `slice_*` functions.

```
gapminder |>
  filter(year == 1952) |>
```

```
pull(continent)
```

```
[1] Asia      Europe  Africa  Africa  Americas Oceania  Europe  Asia
[9] Asia      Europe  Africa  Americas Europe    Africa  Americas Europe
[17] Africa   Africa  Asia     Africa  Americas Africa   Africa  Americas
[25] Asia      Americas Africa  Africa  Africa   Americas Africa   Europe
[33] Americas Europe  Europe  Africa  Americas Americas Africa   Americas
[41] Africa   Africa  Africa  Europe  Europe    Africa  Africa   Europe
[49] Africa   Europe  Americas Africa  Africa   Americas Americas  Asia
[57] Europe   Europe  Asia     Asia     Asia      Asia     Europe   Asia
[65] Europe   Americas Asia     Asia     Africa    Asia     Asia     Asia
[73] Asia     Africa  Africa  Africa  Africa    Africa   Africa   Africa
[81] Africa   Africa  Americas Asia     Europe    Africa   Africa   Asia
[89] Africa   Asia     Europe  Oceania  Americas  Africa   Africa   Europe
[97] Asia     Asia     Americas Americas Americas  Asia     Europe   Europe
[105] Americas Africa  Europe  Africa  Africa    Asia     Africa   Europe
[113] Africa   Asia     Europe  Europe  Africa    Africa   Europe   Asia
[121] Africa   Africa  Europe  Europe  Asia      Asia     Africa   Asia
[129] Africa   Americas Africa  Europe  Africa    Europe   Americas  Americas
[137] Americas Asia     Asia     Asia     Africa    Africa
```

Levels: Africa Americas Asia Europe Oceania

```
gapminder |>
  distinct(continent)
```

```
# A tibble: 5 x 1
  continent
  <fct>
1 Asia
2 Europe
3 Africa
4 Americas
5 Oceania
```

```
gapminder |>
  slice(1:10)
```

```
# A tibble: 10 x 6
  country      continent  year lifeExp      pop gdpPercap
```

	<fct>	<fct>	<int>	<dbl>	<int>	<dbl>
1	Afghanistan	Asia	1952	28.8	8425333	779.
2	Afghanistan	Asia	1957	30.3	9240934	821.
3	Afghanistan	Asia	1962	32.0	10267083	853.
4	Afghanistan	Asia	1967	34.0	11537966	836.
5	Afghanistan	Asia	1972	36.1	13079460	740.
6	Afghanistan	Asia	1977	38.4	14880372	786.
7	Afghanistan	Asia	1982	39.9	12881816	978.
8	Afghanistan	Asia	1987	40.8	13867957	852.
9	Afghanistan	Asia	1992	41.7	16317921	649.
10	Afghanistan	Asia	1997	41.8	22227415	635.

```
gapminder |>
  slice_head(n = 5)
```

```
# A tibble: 5 x 6
  country    continent  year lifeExp      pop gdpPercap
  <fct>      <fct>    <int>  <dbl>    <int>    <dbl>
1 Afghanistan Asia      1952   28.8  8425333    779.
2 Afghanistan Asia      1957   30.3  9240934    821.
3 Afghanistan Asia      1962   32.0 10267083    853.
4 Afghanistan Asia      1967   34.0 11537966    836.
5 Afghanistan Asia      1972   36.1 13079460    740.
```

```
gapminder |>
  slice_tail(n = 5)
```

```
# A tibble: 5 x 6
  country    continent  year lifeExp      pop gdpPercap
  <fct>      <fct>    <int>  <dbl>    <int>    <dbl>
1 Zimbabwe Africa      1987   62.4  9216418    706.
2 Zimbabwe Africa      1992   60.4 10704340    693.
3 Zimbabwe Africa      1997   46.8 11404948    792.
4 Zimbabwe Africa      2002   40.0 11926563    672.
5 Zimbabwe Africa      2007   43.5 12311143    470.
```

```
set.seed(1)
gapminder |>
  slice_sample(n = 10)
```



```
# A tibble: 10 x 6
```

	country	continent	year	lifeExp	pop	gdpPercap
	<fct>	<fct>	<int>	<dbl>	<int>	<dbl>
1	Montenegro	Europe	1992	75.4	621621	7003.
2	Hungary	Europe	1982	69.4	10705535	12546.
3	Benin	Africa	1992	53.9	4981671	1191.
4	Malawi	Africa	1977	43.8	5637246	663.
5	Thailand	Asia	1992	67.3	56667095	4617.
6	El Salvador	Americas	1962	52.3	2747687	3777.
7	China	Asia	2002	72.0	1280400000	3119.
8	Chad	Africa	1977	47.4	4388260	1134.
9	Peru	Americas	2002	69.9	26769436	5909.
10	Senegal	Africa	2002	61.6	10870037	1520.

```
gapminder |>  
  filter(year == 2007) |>  
  slice_max(lifeExp, n = 2)
```

```
# A tibble: 2 x 6
```

	country	continent	year	lifeExp	pop	gdpPercap
	<fct>	<fct>	<int>	<dbl>	<int>	<dbl>
1	Japan	Asia	2007	82.6	127467972	31656.
2	Hong Kong, China	Asia	2007	82.2	6980412	39725.

```
gapminder |>  
  filter(year == 2007) |>  
  slice_min(lifeExp, n = 2)
```

```
# A tibble: 2 x 6
```

	country	continent	year	lifeExp	pop	gdpPercap
	<fct>	<fct>	<int>	<dbl>	<int>	<dbl>
1	Swaziland	Africa	2007	39.6	1133066	4513.
2	Mozambique	Africa	2007	42.1	19951656	824.

```
gapminder |>  
  filter(year == 2007 | year == 1952) |>  
  group_by(year) |>  
  slice_max(lifeExp, n = 2)
```

```
# A tibble: 4 x 6
# Groups:   year [2]
  country      continent year lifeExp      pop gdpPercap
  <fct>        <fct>    <int>   <dbl>    <int>    <dbl>
1 Norway      Europe    1952   72.7   3327728   10095.
2 Iceland     Europe    1952   72.5   147962    7268.
3 Japan       Asia     2007   82.6 127467972   31656.
4 Hong Kong, China Asia     2007   82.2   6980412   39725.
```

```
gapminder_united <- gapminder |>
  unite("country_continent", c(country, continent),
        sep = "_",
        remove = TRUE,
        na.rm = FALSE)

gapminder_united |>
  separate_wider_delim(country_continent,
                        delim = "_",
                        names = c("country", "continent"))
```

```
# A tibble: 1,704 x 6
  country      continent year lifeExp      pop gdpPercap
  <chr>        <chr>    <int>   <dbl>    <int>    <dbl>
1 Afghanistan Asia     1952   28.8   8425333    779.
2 Afghanistan Asia     1957   30.3   9240934    821.
3 Afghanistan Asia     1962   32.0 10267083    853.
4 Afghanistan Asia     1967   34.0 11537966    836.
5 Afghanistan Asia     1972   36.1 13079460    740.
6 Afghanistan Asia     1977   38.4 14880372    786.
7 Afghanistan Asia     1982   39.9 12881816    978.
8 Afghanistan Asia     1987   40.8 13867957    852.
9 Afghanistan Asia     1992   41.7 16317921    649.
10 Afghanistan Asia     1997   41.8 22227415    635.
# i 1,694 more rows
```

3.7 Exercises

Let's work with the `billboard` dataset. In this dataset, each observation is a song. The first three columns (artist, track, and date entered) are variables that describe the song. Then, we

have 76 columns (wk1-wk76) that describe the song's ranking each week. Here, the column names are a variable (the week), and the cell values are another (the ranking).

```
library(tidyverse)
billboard

# A tibble: 317 x 79
  artist      track date.entered  wk1  wk2  wk3  wk4  wk5  wk6  wk7  wk8
  <chr>      <chr> <date>      <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1 2 Pac      Baby~ 2000-02-26    87   82   72   77   87   94   99   NA
2 2Ge+her    The ~ 2000-09-02    91   87   92   NA   NA   NA   NA   NA
3 3 Doors D~ Kryp~ 2000-04-08    81   70   68   67   66   57   54   53
4 3 Doors D~ Loser 2000-10-21    76   76   72   69   67   65   55   59
5 504 Boyz   Wobb~ 2000-04-15    57   34   25   17   17   31   36   49
6 98^0       Give~ 2000-08-19    51   39   34   26   26   19    2    2
7 A*Teens    Danc~ 2000-07-08    97   97   96   95  100   NA   NA   NA
8 Aaliyah    I Do~ 2000-01-29    84   62   51   41   38   35   35   38
9 Aaliyah    Try ~ 2000-03-18    59   53   38   28   21   18   16   14
10 Adams, Yo~ Open~ 2000-08-26    76   76   74   69   68   67   61   58
# i 307 more rows
# i 68 more variables: wk9 <dbl>, wk10 <dbl>, wk11 <dbl>, wk12 <dbl>,
# wk13 <dbl>, wk14 <dbl>, wk15 <dbl>, wk16 <dbl>, wk17 <dbl>, wk18 <dbl>,
# wk19 <dbl>, wk20 <dbl>, wk21 <dbl>, wk22 <dbl>, wk23 <dbl>, wk24 <dbl>,
# wk25 <dbl>, wk26 <dbl>, wk27 <dbl>, wk28 <dbl>, wk29 <dbl>, wk30 <dbl>,
# wk31 <dbl>, wk32 <dbl>, wk33 <dbl>, wk34 <dbl>, wk35 <dbl>, wk36 <dbl>,
# wk37 <dbl>, wk38 <dbl>, wk39 <dbl>, wk40 <dbl>, wk41 <dbl>, wk42 <dbl>, ...
```

a) Apply a transformation to the dataset to leave it in the format below.

```
# A tibble: 24,092 x 5
  artist track      date.entered week  rank
  <chr>  <chr>      <date>      <chr> <dbl>
1 2 Pac  Baby Don't Cry (Keep... 2000-02-26 wk1    87
2 2 Pac  Baby Don't Cry (Keep... 2000-02-26 wk2    82
3 2 Pac  Baby Don't Cry (Keep... 2000-02-26 wk3    72
4 2 Pac  Baby Don't Cry (Keep... 2000-02-26 wk4    77
5 2 Pac  Baby Don't Cry (Keep... 2000-02-26 wk5    87
6 2 Pac  Baby Don't Cry (Keep... 2000-02-26 wk6    94
7 2 Pac  Baby Don't Cry (Keep... 2000-02-26 wk7    99
8 2 Pac  Baby Don't Cry (Keep... 2000-02-26 wk8    NA
9 2 Pac  Baby Don't Cry (Keep... 2000-02-26 wk9    NA
```

```
10 2 Pac Baby Don't Cry (Keep... 2000-02-26 wk10 NA
# i 24,082 more rows
```

b) Observe the result of item **a)**. What happens if a song is in the top 100 for less than 76 weeks? Take the song “Baby Don’t Cry” by 2 Pac, for example. The output above suggests that it was in the top 100 for only 7 weeks, and all remaining weeks are filled with missing values (NA). These NAs actually do not represent unknown observations; they were forced to exist by the structure of the dataset. Change the code used in **a)** to remove these NAs. Answer: How many rows are left? (Hint: see the documentation of the `pivot_longer` function.)

c) You may have noticed that in the result of item **a)**, the type of the `week` column is character. Perform the appropriate transformation to obtain a column with numerical values.

d) Which song stayed in the top 100 of the Billboard in 2000 for the most weeks? How many weeks did this song appear in the ranking? And which song stayed in the ranking for the least time?

e) Which song stayed exactly 10 weeks in the top 100 of the Billboard in 2000? If there is more than one song in this condition, consider the one that first entered the ranking.

4 Data Visualization

Data visualization plays a fundamental role in understanding and communicating complex information. By transforming numbers and statistics into charts, tables, and other visual representations, we can identify patterns, trends, and insights that may not be immediately apparent in raw datasets. Additionally, data visualization facilitates the communication of results and findings to a broader audience, making complex information more accessible and understandable.

4.1 Grammar of Graphics

It is based on the grammar of graphics (*grammar of graphics* - GG), see Wilkinson (2012). The grammar of graphics is a set of principles and concepts that describe the structure and rules for creating charts consistently and effectively. GG is a framework for data visualization that breaks down each component of a chart into individual elements, creating distinct layers. Using the GG system, we can build charts step by step to achieve flexible and customizable results. Each aspect of the chart, such as points, lines, colors, and scales, is treated as a separate piece, allowing detailed control over the appearance and content of the final chart.

4.2 The ggplot Package

The most well-known data visualization package in R is `ggplot2`, which is based on the grammar of graphics. `ggplot2` allows the creation of a wide variety of charts, including scatter plots, line plots, bar charts, histograms, and more, in a simple and flexible way. With `ggplot2`, you can customize basically all aspects of the chart, from the shape and color of the points to the axis scales and the background appearance.

To exemplify the use of `ggplot`, let's consider the `gapminder` dataset, Rosling (2012).

4.2.1 Data

The `ggplot()` function initializes a `ggplot2` chart and defines the data to be used.

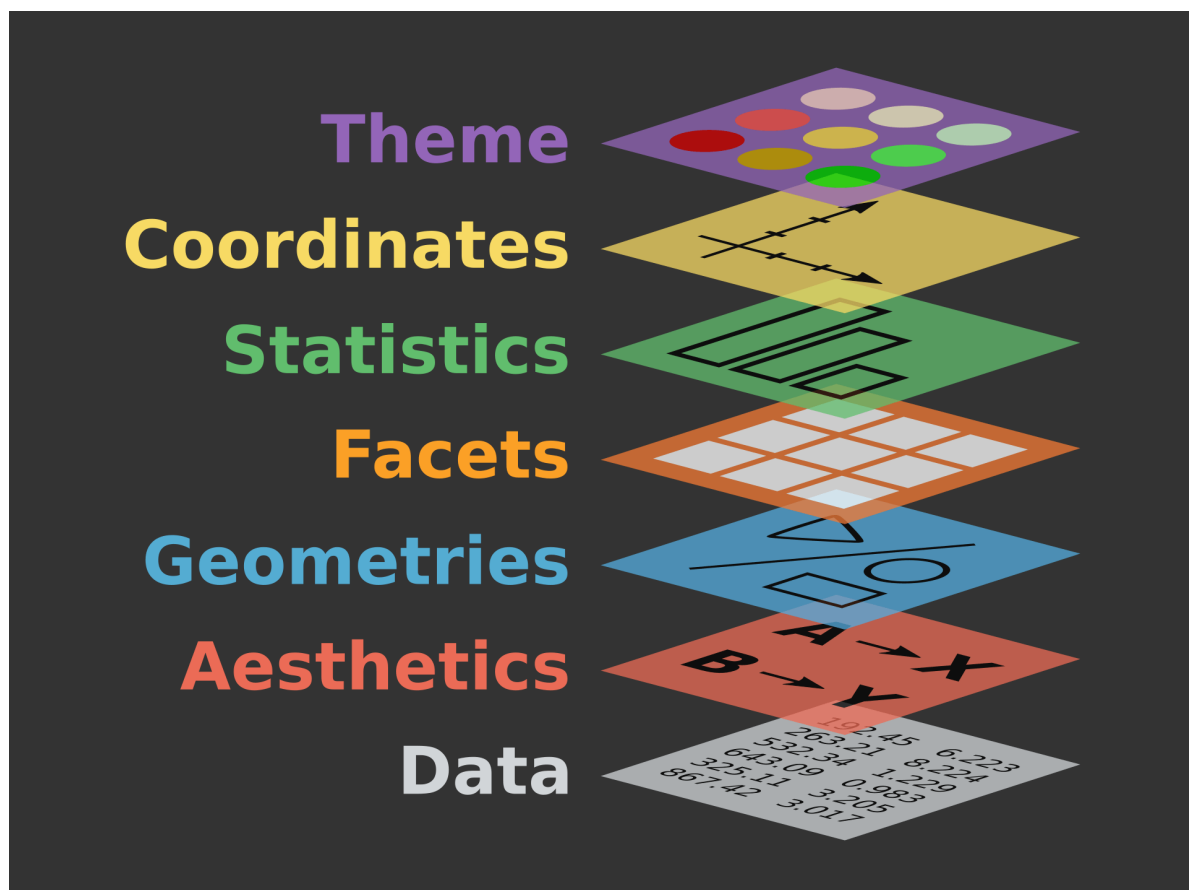
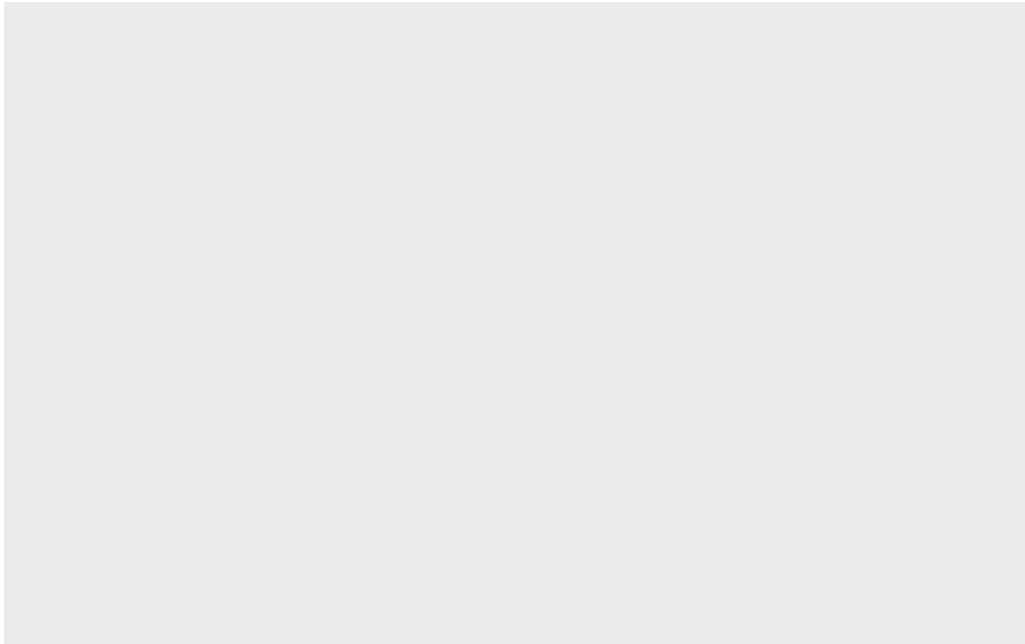


Figure 4.1: Figure from Wilkinson (2012).

```
library(tidyverse)
library(gapminder)

gapminder_2007 <- gapminder %>%
  filter(year == 2007)

ggplot(data = gapminder_2007)
```



i The generated plot is also not wrong!

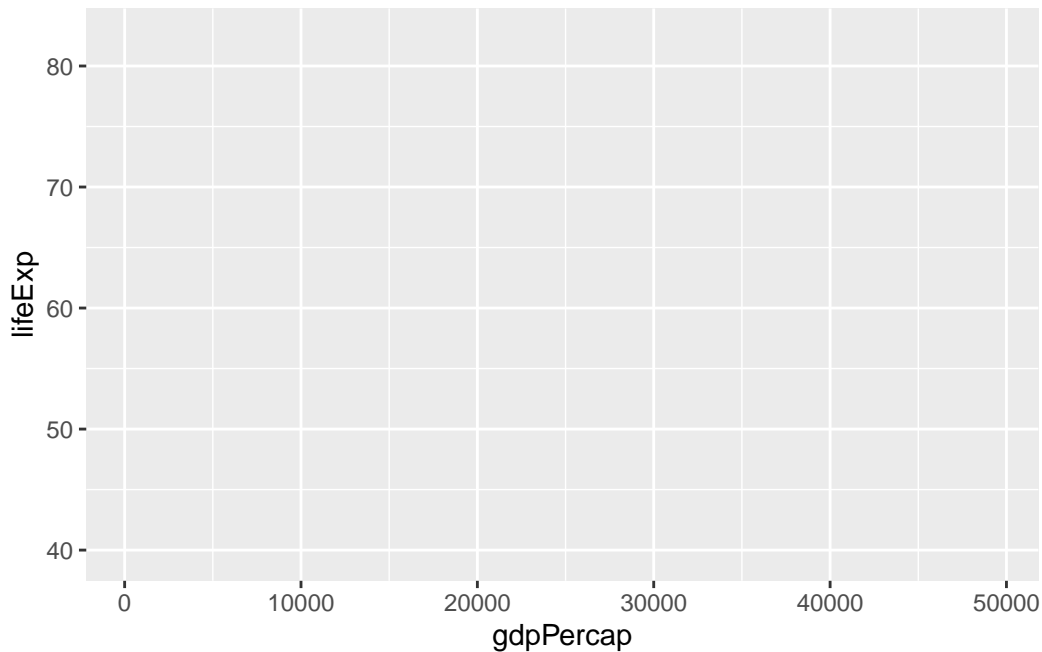
This code creates a chart using the `gapminder` data for the year 2007 only. That's all. There are no extra instructions about what to display on each axis (chart aesthetics).

4.2.2 Aesthetics

The aesthetic mapping (`aes`) in `ggplot2` is a function that allows linking variables from a dataset to the visual properties of a chart, such as color, shape, size, and position. Through aesthetic mapping, we can control how the data is visually represented in the chart.

For example, when creating a scatter plot, we can map the `x` (horizontal) variable and the `y` (vertical) variable from the dataset to the chart coordinates.

```
ggplot(data = gapminder_2007,  
       mapping = aes(x = gdpPercap, y = lifeExp))
```



i The generated plot is also not wrong!

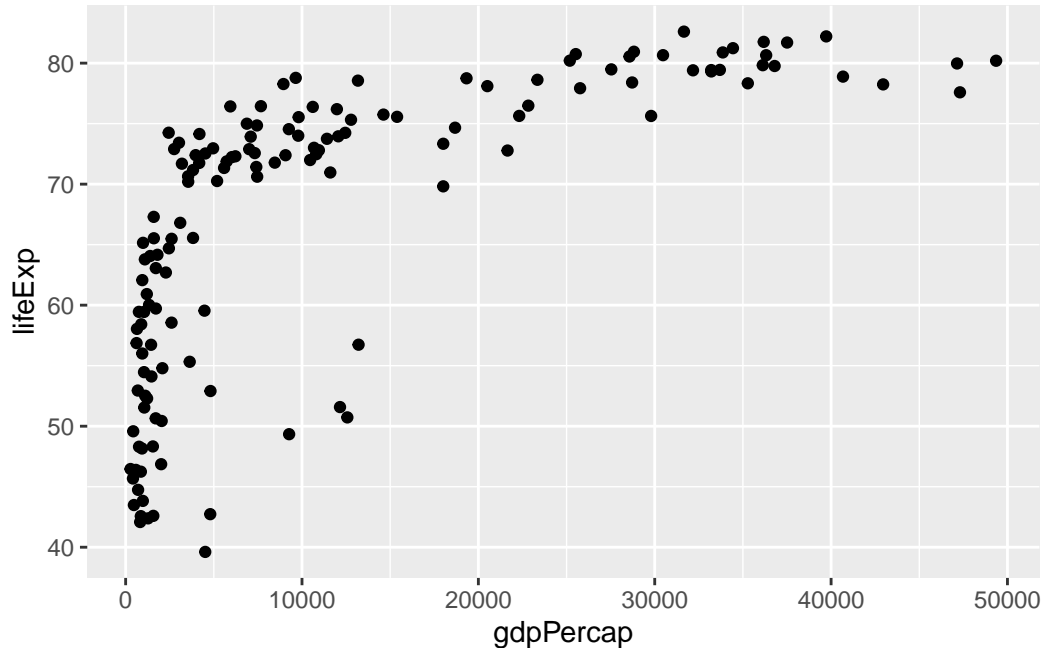
This code defines a scatter plot using the gapminder data for the year 2007 only. The x-axis represents GDP per capita (gdpPercap), and the y-axis represents life expectancy (lifeExp). That's all. There are no extra instructions in the code snippet about the format (geometry) that should be used to display the data.

4.2.3 Geometry

Geometry refers to the visual elements that make up a chart, such as points, lines, bars, and areas. Each chart type has its corresponding geometry, specified by the `geom_*` function followed by the desired geometry type.

For example, to create a scatter plot, we use the `geom_point()` geometry, while for creating a bar chart, we use the `geom_bar()` geometry.


```
ggplot(data = gapminder_2007,
       mapping = aes(x = gdpPercap, y = lifeExp)) +
  geom_point()
```

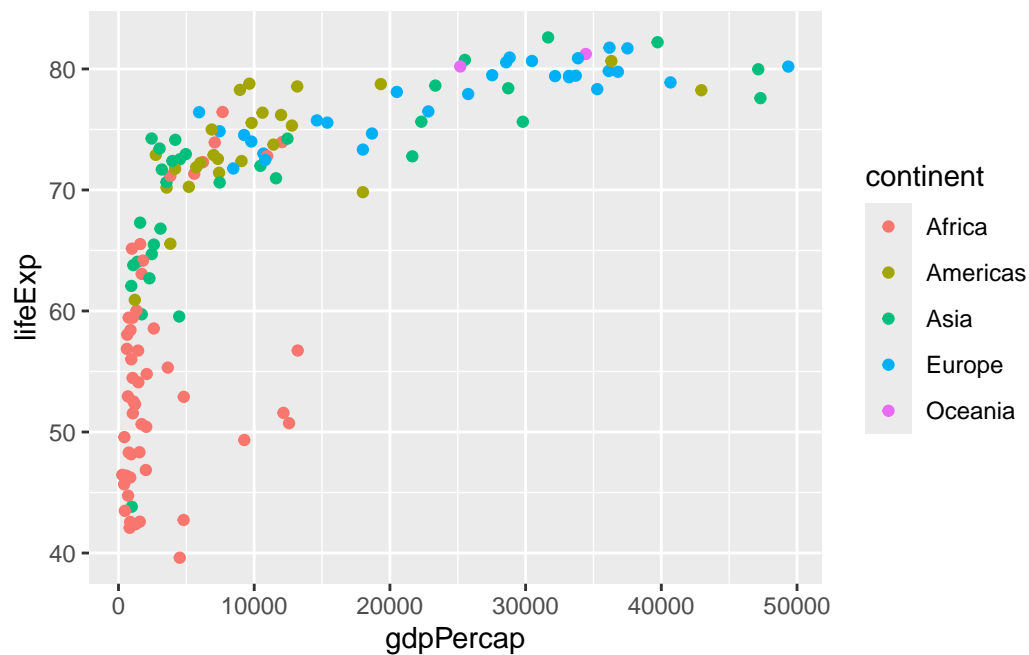


Each geometry has its specific parameters that can be adjusted to customize the appearance of the chart, such as color, size, fill, and transparency.

💡 Adding color aesthetics to the chart

To color each point in the scatter plot according to the continent, just add `color = continent` to the aesthetic mapping.

```
ggplot(data = gapminder_2007,
       mapping = aes(x = gdpPercap, y = lifeExp, color = continent)) +
  geom_point()
```



4.2.3.1 Other Geometries

Below are examples of other possible geometries. Some changes were made to how the `ggplot` functions are used. Purposefully, no descriptive text was added as these changes should be easy to interpret.

`geom_line()`:

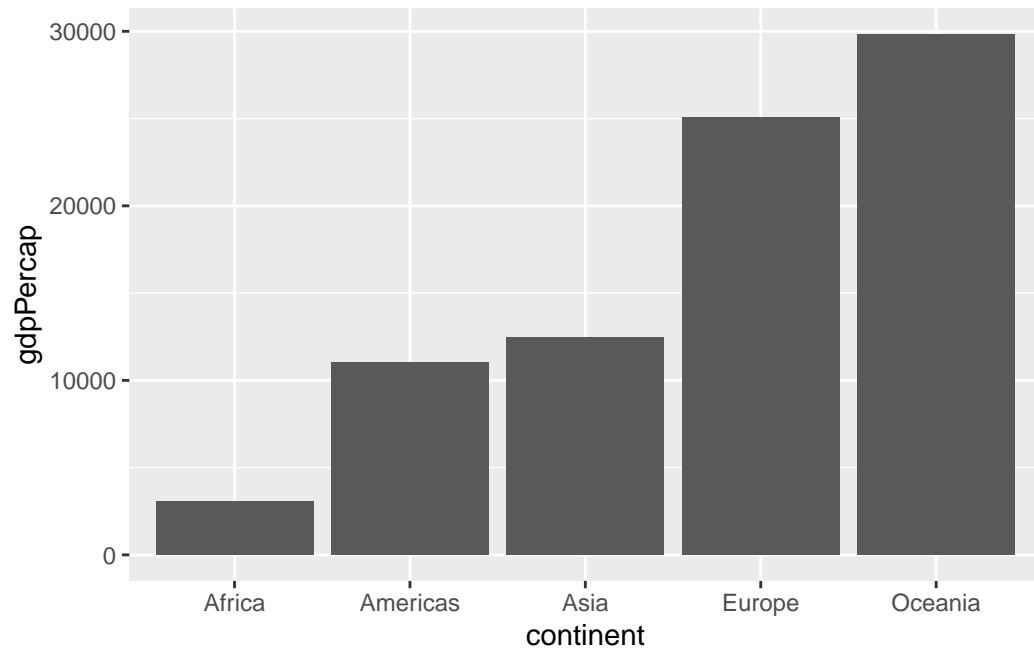
```
# Filter data for Brazil
dados_pais <- gapminder %>%
  filter(country == "Brazil")

# Create line plot
ggplot(data = gapminder %>%
  filter(country == "Brazil"),
  aes(x = year, y = lifeExp)) +
  geom_line()
```



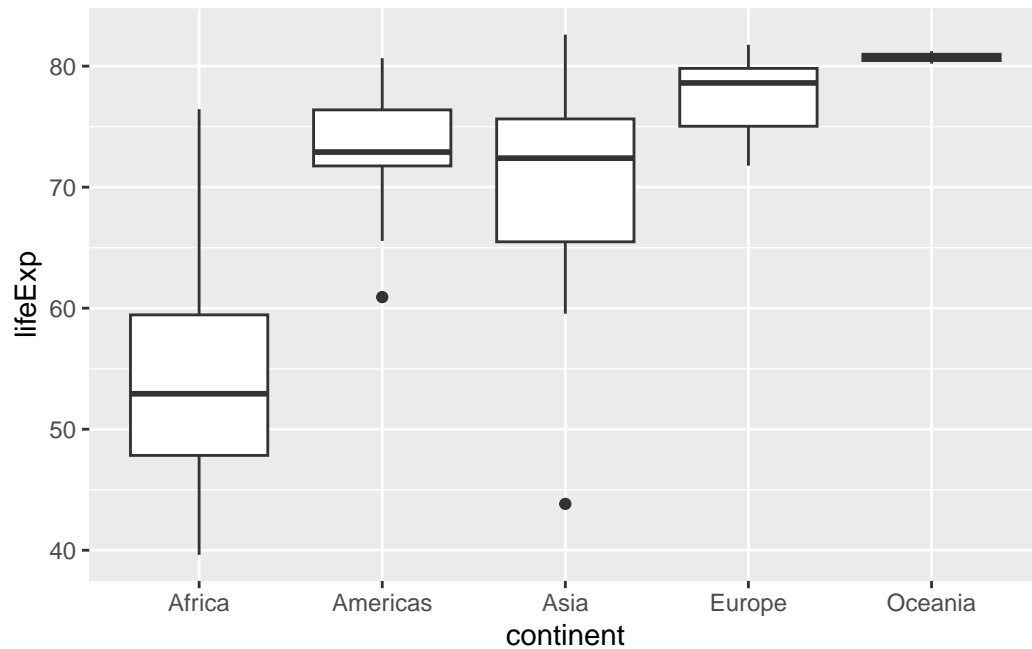
`geom_bar()`:

```
# Create bar plot
ggplot(data = gapminder_2007, aes(x = continent, y = gdpPercap)) +
  geom_bar(stat = "summary", fun = "mean")
```



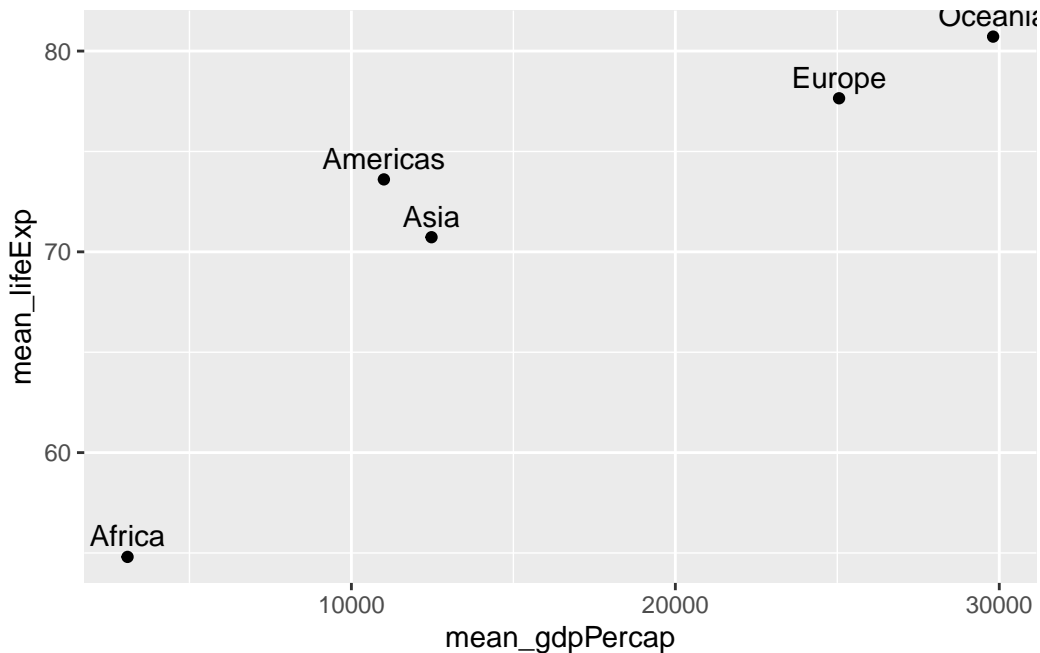
`geom_boxplot()`:

```
ggplot(data = gapminder_2007, aes(x = continent, y = lifeExp)) +  
  geom_boxplot()
```



geom_text():

```
gapminder_2007 %>%
  group_by(continent) %>%
  summarise(mean_lifeExp = mean(lifeExp),
            mean_gdpPercap = mean(gdpPercap)) %>%
  ggplot(aes(x = mean_gdpPercap, y = mean_lifeExp, label = continent)) +
  geom_point() +
  geom_text(vjust = -0.5, hjust = 0.5)
```



Note that we are combining two geometries in a single plot (point and text). Text labels are added to points using the `geom_text()` geometry, with the parameters `vjust` and `hjust` defining the vertical and horizontal position of the text, respectively:

- The `vjust` parameter adjusts the vertical alignment of the text relative to the point. A negative value (-0.5, for example) moves the text above the point, while a positive value moves it below the point.
- The `hjust` parameter adjusts the horizontal alignment of the text relative to the point. A value of 0.5 centers the text horizontally relative to the point.

4.2.4 Facets

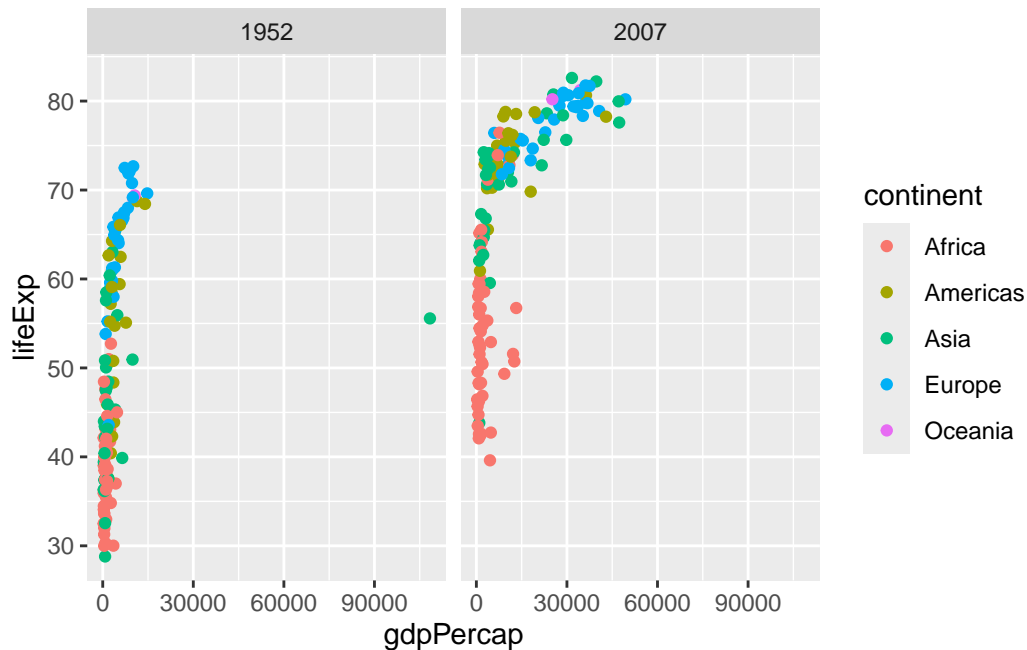
Facets refer to the ability to split a plot into multiple visualizations based on one or more categorical variables. This allows comparing relationships between variables in different data segments.

Facets are added using the `facet_wrap()` function to create a matrix of panels based on a categorical variable or `facet_grid()` to create a panel grid based on two categorical variables.

For example, we can use facets to create separate scatter plots for two distinct years, allowing comparisons of the relationships between GDP per capita and life expectancy in those two years.

```
gapminder_anos <- gapminder %>%
  filter(year == 1952 | year == 2007)

ggplot(data = gapminder_anos,
       mapping = aes(x = gdpPercap, y = lifeExp, color = continent)) +
  geom_point() +
  facet_wrap(~year)
```



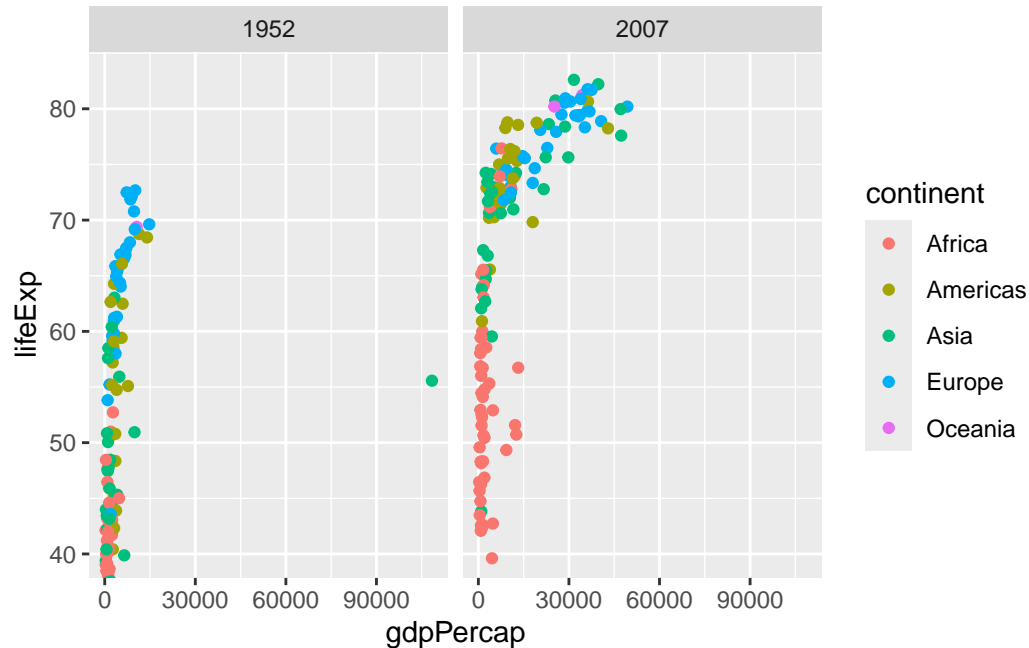
4.2.5 Coordinates

Coordinates in ggplot2 determine how data is mapped in a graphic space. This includes the scales of the x and y axes, as well as any transformation or adjustment applied to the data. Coordinates affect the overall appearance of the chart, including its orientation, proportion, and scale.

To set limits on the x and y axes, we can use the `coord_cartesian()` function to control which value ranges are displayed on the chart. This is useful when we want to focus on a specific part of the data or prevent outliers from influencing the axis scales.

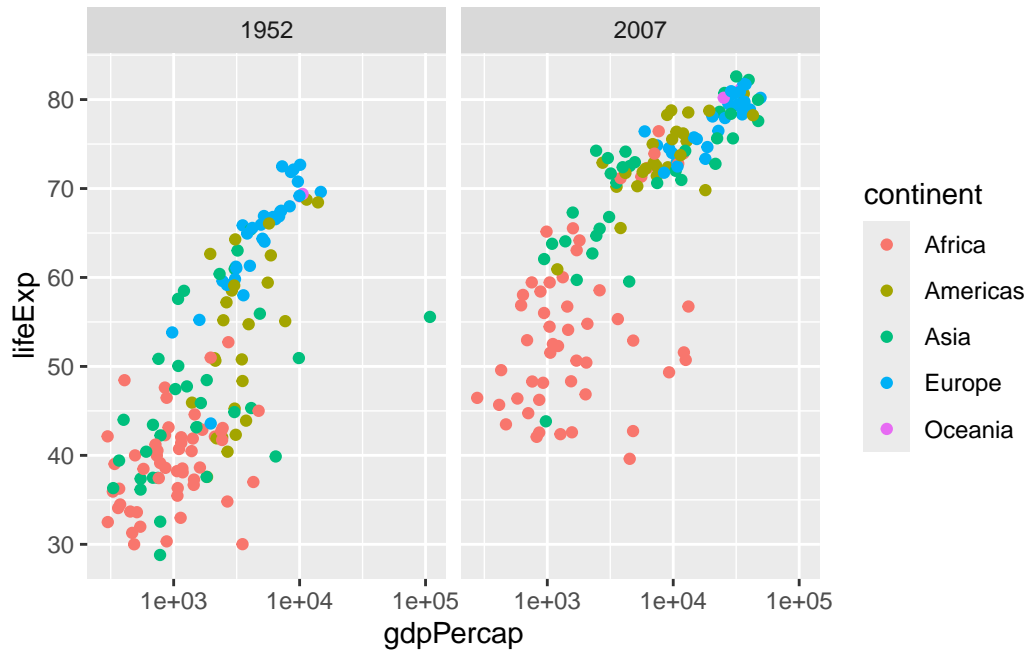
```
ggplot(data = gapminder_anos,
       mapping = aes(x = gdpPercap, y = lifeExp, color = continent)) +
```

```
geom_point() +
facet_wrap(~year) +
coord_cartesian(ylim = c(40, 83))
```



It is possible to apply a logarithmic scale to the axes as well. This is useful when the data has a wide range of values and we are interested in highlighting differences across a broad range of values, such as in income or GDP data. To do this, simply use the `scale*_log10()` function:

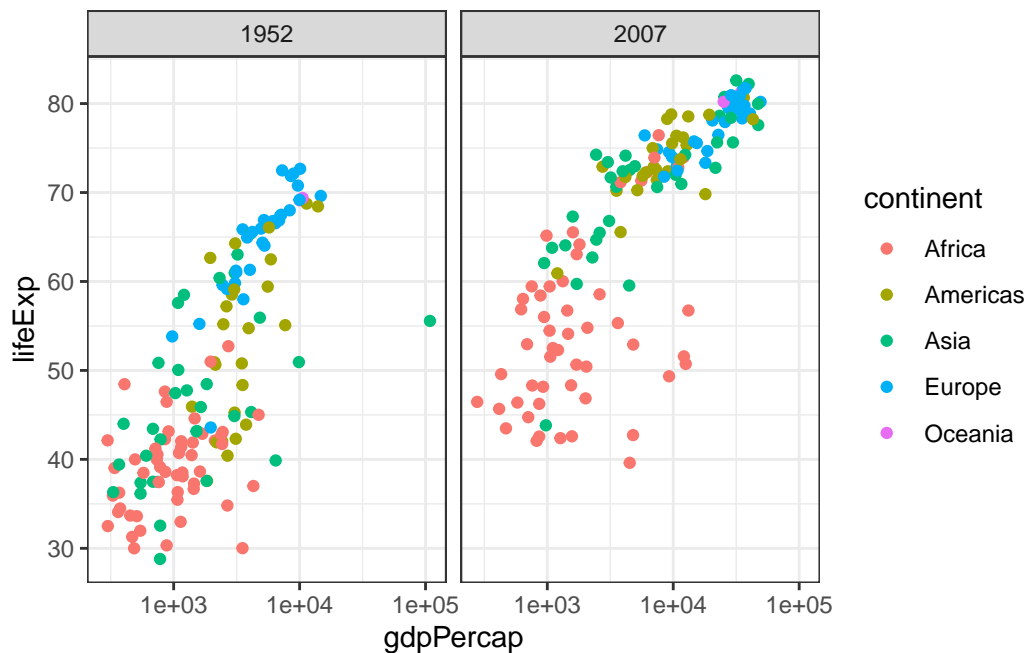
```
ggplot(data = gapminder_anos,
       mapping = aes(x = gdpPercap, y = lifeExp, color = continent)) +
geom_point() +
facet_wrap(~year) +
scale_x_log10()
```

4.2.6 Themes

Themes control the visual aspects of charts, such as titles, legends, axes, and background colors. Predefined themes, such as `theme_bw()`, `theme_minimal()`, and `theme_classic()`, offer consistent visual styles that can be applied to charts for a specific appearance.

```
ggplot(data = gapminder_anos,
       mapping = aes(x = gdpPercap, y = lifeExp, color = continent)) +
  geom_point() +
  facet_wrap(~year) +
  scale_x_log10() +
  theme_bw()
```

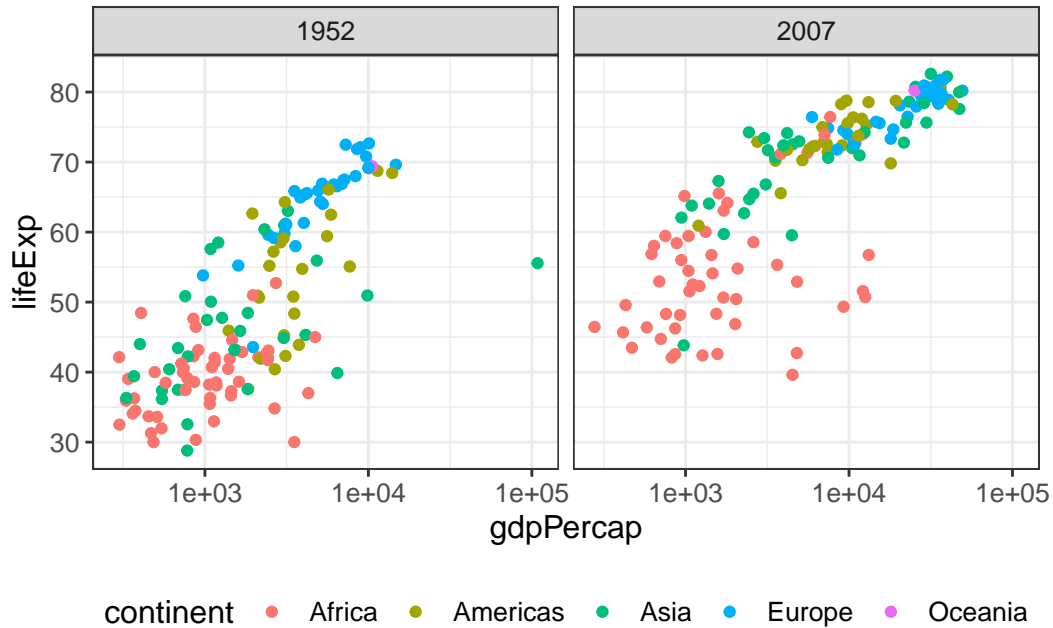


💡 Try it too

Delete the last line from the example above, type `theme_`, press the tab key, and experiment with the different predefined themes in `ggplot`.

Additionally, we can define virtually all aspects of the chart. For example, to move the legend position to the bottom, we can use the function `theme(legend.position = "bottom")`. To change the font size, we can use the function `theme(text = element_text(size = 12))`. These settings can be combined in a single call to the `theme()` function, as shown below.

```
ggplot(data = gapminder_anos,
       mapping = aes(x = gdpPercap, y = lifeExp, color = continent)) +
  geom_point() +
  facet_wrap(~year) +
  scale_x_log10() +
  theme_bw() +
  theme(legend.position = "bottom",
        text = element_text(size = 12))
```

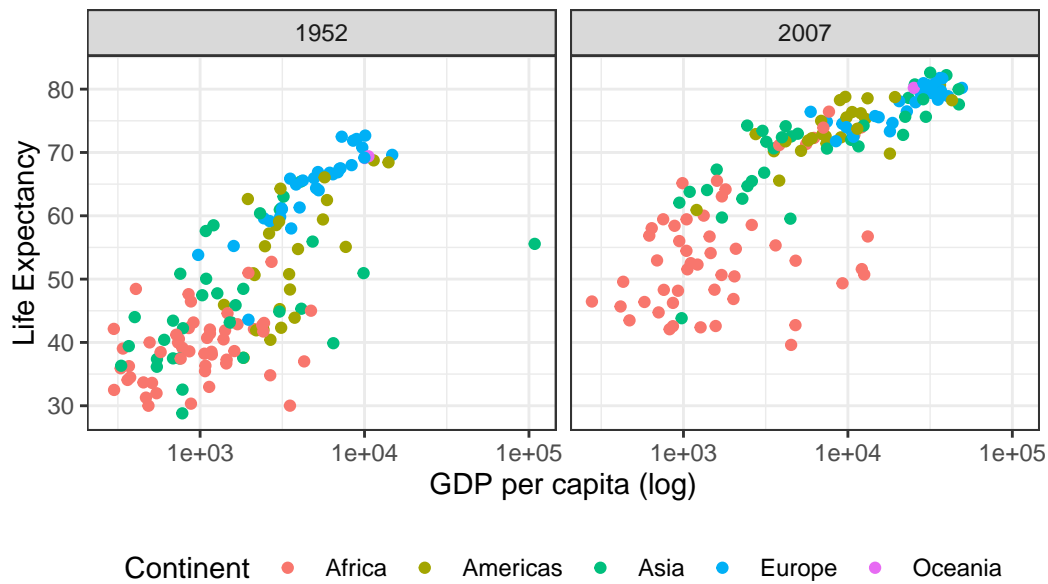


4.2.7 Customization and Styling of Graphs

The `labs()` function is responsible for customizing labels and titles in plots. In the example below, we rename the x and y axes and assign a more descriptive name to the color legend, which in this case represents the continent.

```
ggplot(data = gapminder_anos,
       mapping = aes(x = gdpPercap, y = lifeExp, color = continent)) +
  geom_point() +
  facet_wrap(~year) +
  scale_x_log10() +
  labs(x = "GDP per capita (log)",
       y = "Life Expectancy",
       color = "Continent",
       title = "Relationship between GDP per capita and Life Expectancy") +
  theme_bw() +
  theme(legend.position = "bottom")
```

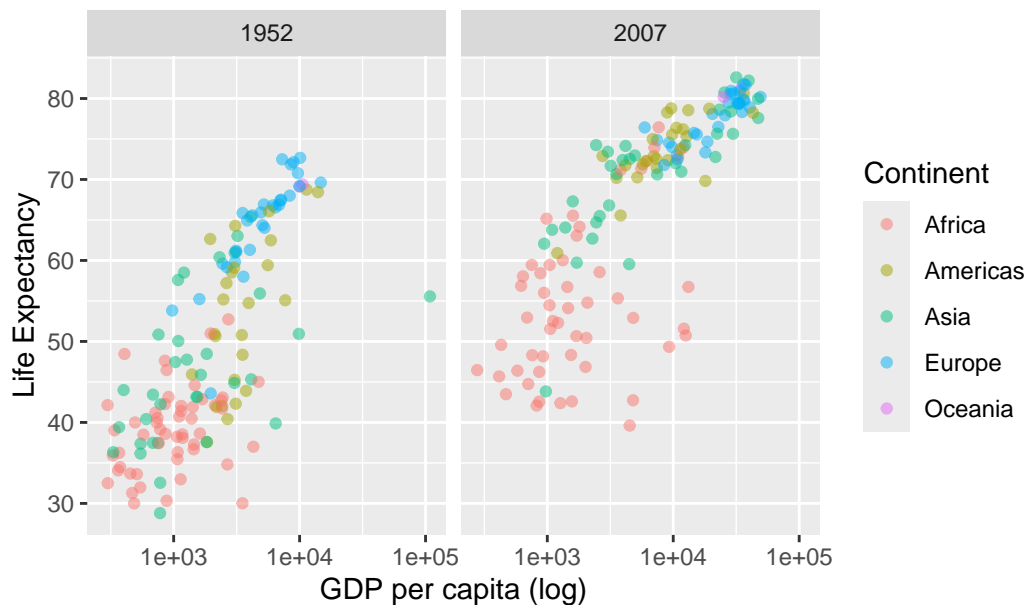
Relationship between GDP per capita and Life Expectancy



The `alpha` parameter controls the opacity of geometric elements, ranging from 0 to 1. For example, `geom_point(alpha = 0.5)` makes the points semi-transparent, which can be useful for visualizing data overlaps in a scatter plot.

```
ggplot(data = gapminder_anos,
       mapping = aes(x = gdpPercap, y = lifeExp, color = continent)) +
  geom_point(alpha = 0.5) +
  facet_wrap(~year) +
  scale_x_log10() +
  labs(x = "GDP per capita (log)",
       y = "Life Expectancy",
       color = "Continent",
       title = "Relationship between GDP per capita and Life Expectancy")
```

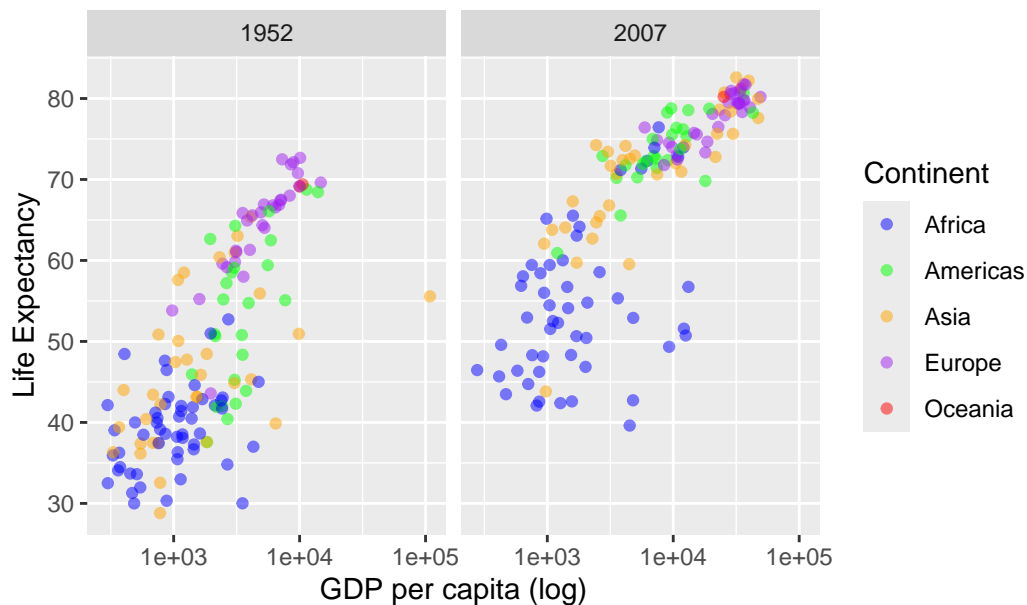
Relationship between GDP per capita and Life Expectancy



To choose specific colors for the levels of a categorical variable, we can use the `scale_color_manual()` function to manually assign colors to each level of the variable.

```
ggplot(data = gapminder_anos,
       mapping = aes(x = gdpPercap, y = lifeExp, color = continent)) +
  geom_point(alpha = 0.5) +
  facet_wrap(~year) +
  scale_x_log10() +
  scale_color_manual(values = c("blue", "green", "orange", "purple", "red"))+
  labs(x = "GDP per capita (log)",
       y = "Life Expectancy",
       color = "Continent",
       title = "Relationship between GDP per capita and Life Expectancy")
```

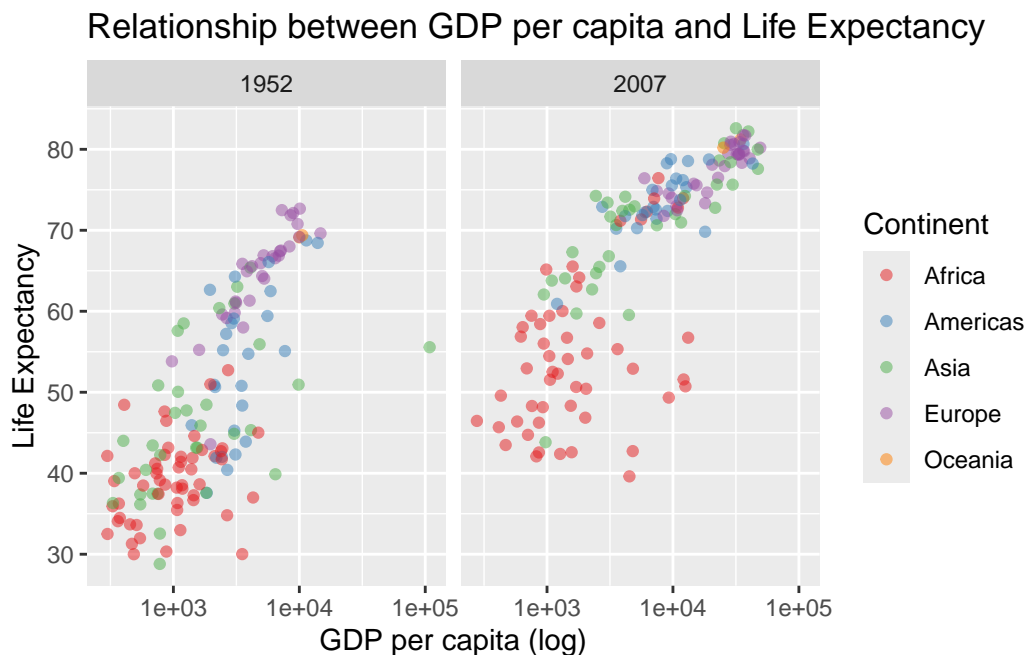
Relationship between GDP per capita and Life Expectancy



💡 Other Color Palettes

You can use color palettes from the `RColorBrewer` package using the `scale_color_brewer()` function.

```
ggplot(data = gapminder_anos,
       mapping = aes(x = gdpPercap, y = lifeExp, color = continent)) +
  geom_point(alpha = 0.5) +
  facet_wrap(~year) +
  scale_x_log10() +
  scale_color_brewer(palette = "Set1")+
  labs(x = "GDP per capita (log)",
       y = "Life Expectancy",
       color = "Continent",
       title = "Relationship between GDP per capita and Life Expectancy")
```



The advantage of using RColorBrewer color palettes is that they are carefully designed to be perceptually distinct and suitable for representing different groups or categories in plots. This means that the colors in a palette are more easily distinguishable from each other, even when printed in black and white or viewed by people with visual impairments. See all available palettes [here](#).

4.3 Additional Packages

There are some extra packages that work as extensions of `ggplot2`. We present some in this section.

4.3.1 The `patchwork` Package

The `patchwork` package is used to combine multiple `ggplot2` plots into a single visualization. It allows you to create flexible and complex layouts, adding, organizing, and adjusting individual plots.

```
# Install the patchwork package (only if not already installed)
install.packages("patchwork")
```

After loading the package, you can use the + operator to combine ggplot2 plots into a single visualization.

Here is a simple example creating two separate plots and then combining them using patchwork:

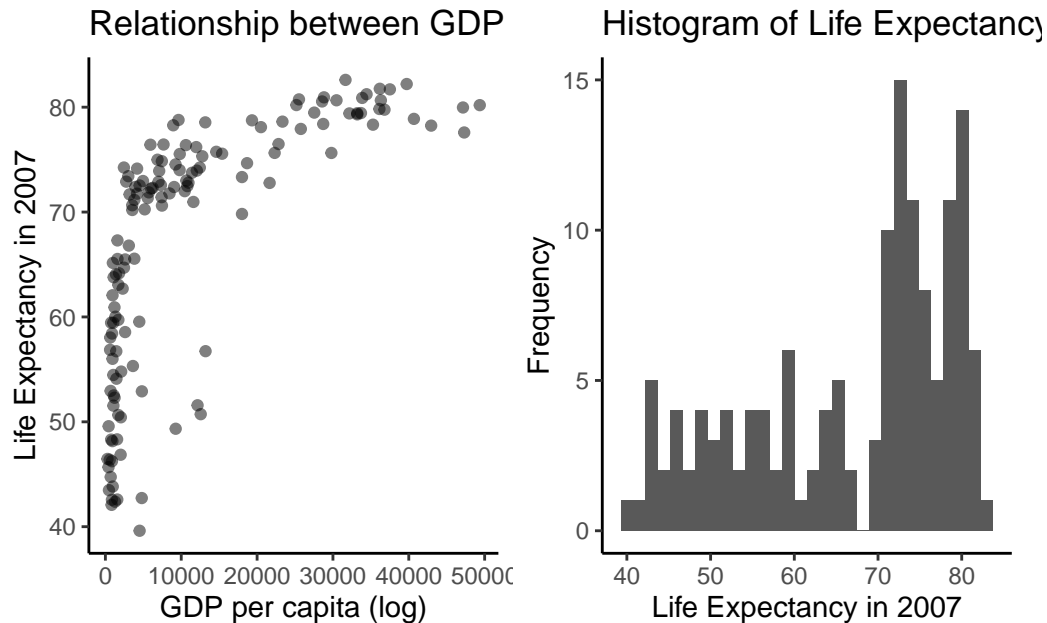
```
library(patchwork)
```

Warning: package 'patchwork' was built under R version 4.2.3

```
plot1 <- ggplot(data = gapminder_2007,
  mapping = aes(x = gdpPercap, y = lifeExp)) +
  geom_point(alpha = 0.5) +
  labs(x = "GDP per capita (log)",
    y = "Life Expectancy in 2007",
    title = "Relationship between GDP per capita and Life Expectancy") +
  theme_classic()

plot2 <- ggplot(data = gapminder_2007,
  mapping = aes(lifeExp)) +
  geom_histogram() +
  labs(title = "Histogram of Life Expectancy",
    x = "Life Expectancy in 2007",
    y = "Frequency") +
  theme_classic()

plot1 + plot2
```

4.3.2 The ggthemes Package

The `ggthemes` package is an extension of `ggplot2` that provides a variety of pre-defined themes to customize the appearance of plots. See the [documentation here](#).

```
# Install the ggthemes package (only if not already installed)
install.packages("ggthemes")
```

After loading the package, you can apply any of the available themes to your `ggplot2` plots using the `theme_*()` function. In the example below, three versions of the same plot are created with different themes. The `+` and `/` operators are used to define how the plots will be displayed.

```
# Load the ggthemes package
library(ggthemes)
```

Warning: package 'ggthemes' was built under R version 4.2.3

```
plot0 <- ggplot(data = gapminder_2007,
  mapping = aes(x = gdpPercap, y = lifeExp)) +
  geom_point(alpha = 0.5) +
```

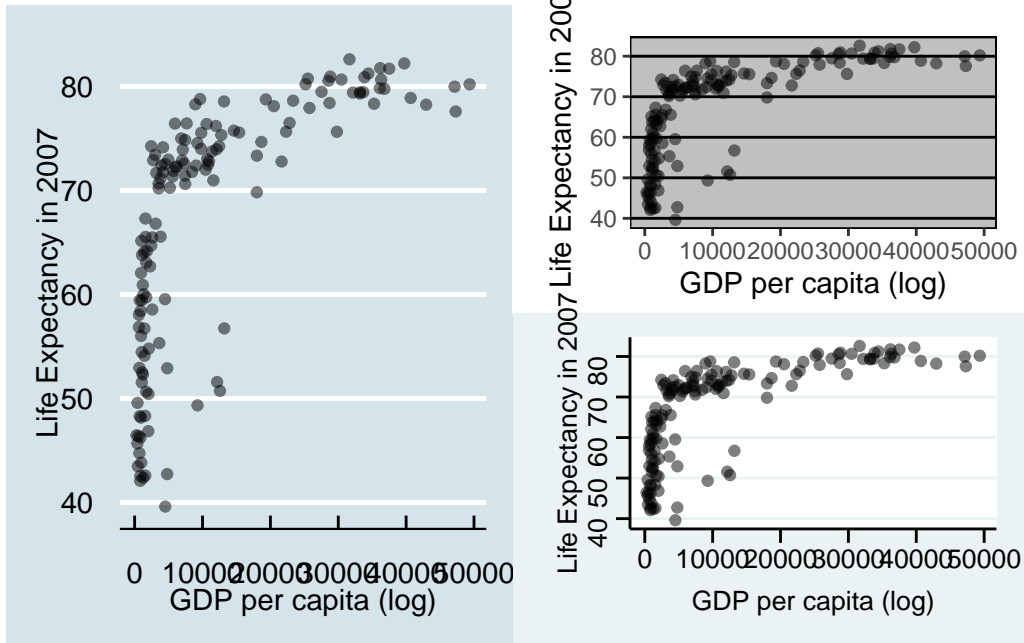
```

labs(x = "GDP per capita (log)",
     y = "Life Expectancy in 2007")

plot1 <- plot0 + theme_economist()
plot2 <- plot0 + theme_excel()
plot3 <- plot0 + theme_stata()

plot1 + (plot2 / plot3)

```



4.3.3 The plotly Package

The `plotly` package offers features to create interactive plots. To add interactivity to the provided plot, we can use the `ggplotly()` function to convert a plot created with `ggplot2` into an interactive plot. Use the command `install.packages("plotly")` if you don't have the package installed.

```
library(plotly)
```

Warning: package 'plotly' was built under R version 4.2.3

```
grafico <- ggplot(data = gapminder_anos,
                 mapping = aes(x = gdpPercap, y = lifeExp,
                              color = continent, text = country)) +
  geom_point(alpha = 0.5) +
  facet_wrap(~year) +
  scale_x_log10() +
  labs(x = "GDP per capita (log)",
       y = "Life Expectancy",
       color = "Continent",
       title = "Relationship between GDP per capita and Life Expectancy")

ggplotly(grafico)
```

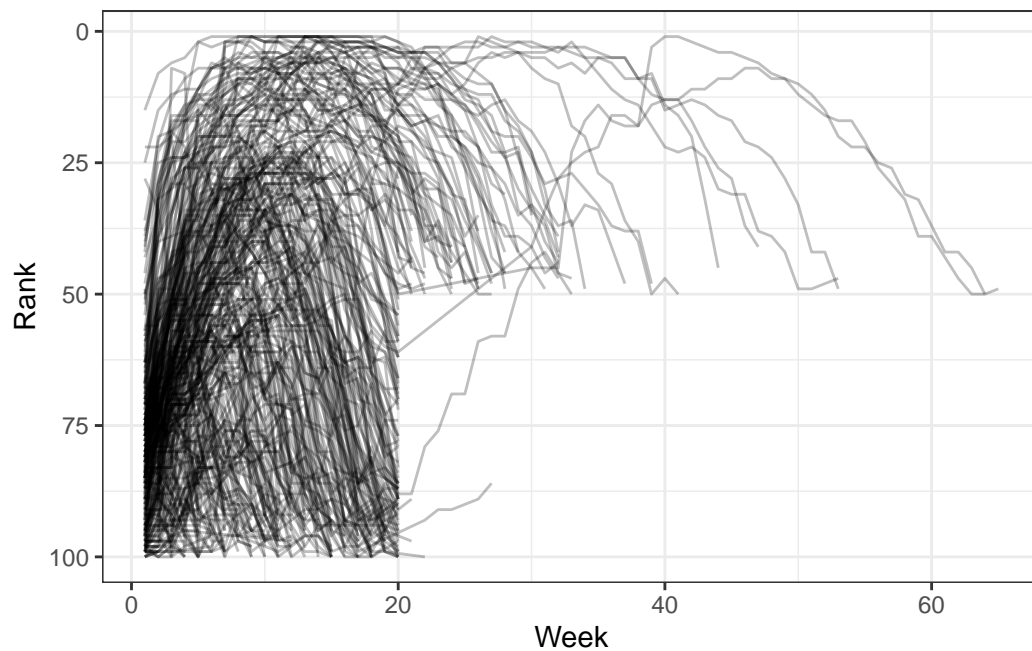
4.4 Extra Tips

When exploring different types of plots to visualize your data, the website [Data to Viz](#) can be a valuable tool. It provides a complete gallery of plot types and offers guidance on when and how to use each one. Additionally, the website provides specific examples of how to create these plots using different libraries, such as ggplot2 in R and matplotlib in Python.

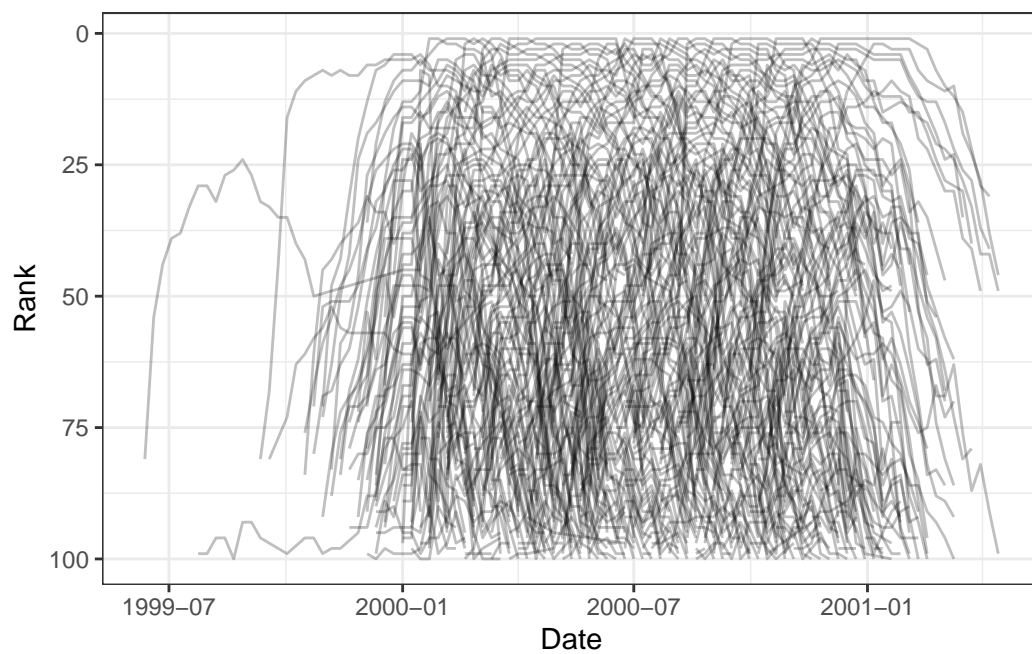
4.5 Exercises

1. Let's use the Billboard data presented in Section 3.7. Your task is to reproduce the plots below using data processing techniques with `dplyr` and data visualization with `ggplot`.

a) The plot below shows the history of each song in the ranking over the weeks.

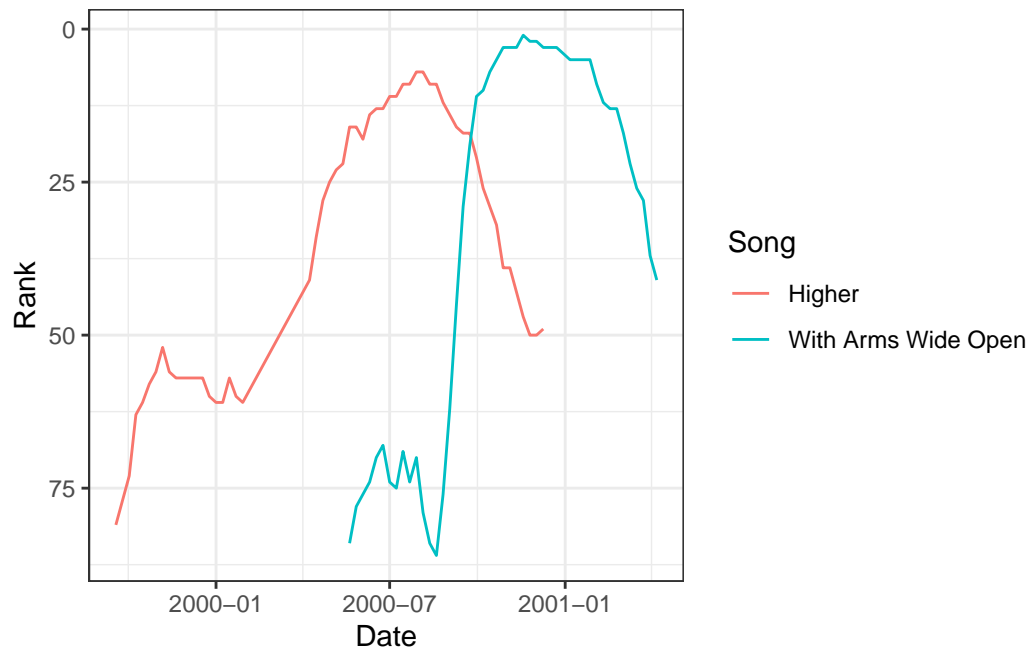


b) The plot below is a small modification of the one presented in item a); the x-axis shows the date the song entered the ranking.

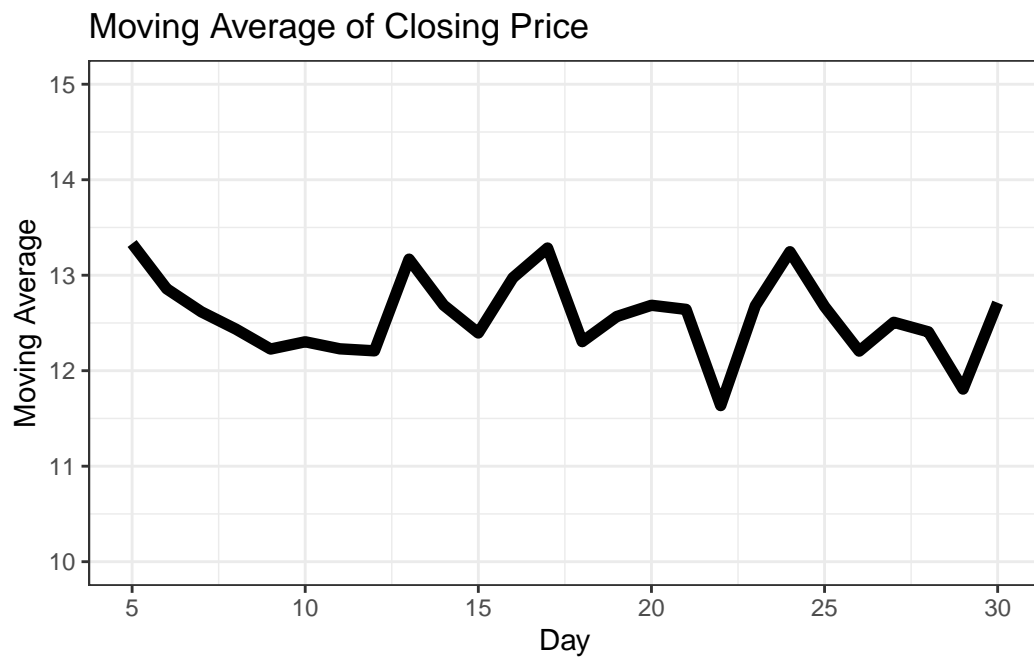


c) The plot below shows the ranking history of two songs: “Higher” and “With Arms Wide

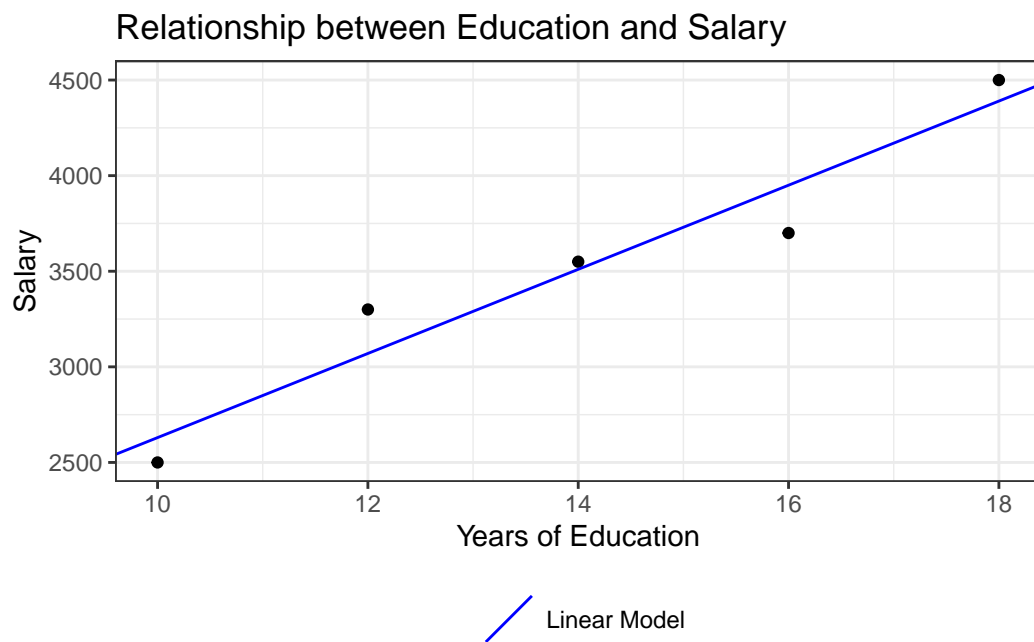
Open.”



2. Use the data generated in Section 2.2.1 representing the closing price of a stock and reproduce the plot below, showing the evolution of the moving average over time.



3. (Challenge) Reproduce the scatter plot presented in Section 2.3.



4. Reproduce the figure presented in Section 3.5.6.

Part II

Learning Python

5 Fundamentos de Python

Notice!

Chapter to be translated!

Atenção!!!

A partir de agora, todo o código apresentado neste livro está na linguagem Python.

A fonte para a construção deste material é McKinney (2022).

5.1 Instalação

Siga os passos abaixo para realizar a instalação do Anaconda e do JupyterLab:

1. Baixe e instale o Anaconda a partir do [site oficial](#). Siga as instruções de instalação para o seu sistema operacional específico.
2. Após instalar o Anaconda, abra o Anaconda Navigator e crie um novo ambiente virtual. Navegue até a seção “Environments” e clique em “Create” para adicionar um novo ambiente. Dê um nome ao ambiente e escolha a versão do Python que deseja usar.
3. Após criar o ambiente virtual, ative-o clicando no ambiente virtual recém-criado na lista de ambientes e selecionando “Open Terminal”. No terminal, digite o comando `conda activate nome_do_seu_ambiente` (substitua `nome_do_seu_ambiente` pelo nome do ambiente que você criou).
4. Com o ambiente virtual ativado, instale o JupyterLab digitando `conda install jupyterlab` no terminal.
5. Depois de instalar o JupyterLab, execute-o digitando `jupyter lab` no terminal ou através da interface do Anaconda. Isso abrirá o JupyterLab no seu navegador padrão.
6. No JupyterLab, você pode criar um novo notebook Python clicando no ícone “+” na barra lateral esquerda e selecionando “Python 3” sob o cabeçalho “Notebook”.

5.2 Tipos de dados fundamentais

Em Python, os tipos de dados fundamentais incluem `integer`, `float`, `string` e `boolean`:

- Integers são números inteiros, como 1, 2, -3, etc. - Floats são números decimais, como 3.14, -0.5, etc.
- Strings são sequências de caracteres, como “hello”, “world”, “python”, etc.
- Booleans são valores lógicos que representam verdadeiro (`True`) ou falso (`False`).

Esses tipos de dados são os blocos básicos para representar diferentes tipos de informações em Python, e são amplamente utilizados em programação para realizar operações e manipulações de dados.

5.2.1 O tipo de dado inteiro

Um tipo de dados inteiro (`integer`) em Python representa números inteiros, ou seja, números sem casas decimais. Por exemplo, 5, -10 e 0 são todos exemplos de números inteiros. No Python, os inteiros são representados pela classe `int`. Nos exemplos abaixo realizamos operações básicas com números inteiros.

```
2 + 2
```

4

Na primeira linha, calculamos a soma de 2 com 2.

```
quantidade = 200  
print(quantidade)
```

200

```
type(quantidade)
```

```
<class 'int'>
```

Acima, atribuímos o valor 200 à variável `quantidade` e a imprimimos usando `print(quantidade)`. Por fim, verificamos o tipo de dado da variável `quantidade` com `type(quantidade)`, que retorna `<class 'int'>`, indicando que é um número inteiro.

5.2.2 O tipo de dado ponto flutuante

Um tipo de dado ponto flutuante (`float`) em Python representa números decimais, ou seja, números que podem ter uma parte fracionária. Por exemplo, 3.14, -0.001, e 2.71828 são todos floats. Em Python, os floats são representados pela classe `float`.

```
1.75 + 2**3
```

9.75

```
taxa_juros = 1.25  
  
print(taxa_juros)
```

1.25

```
type(taxa_juros)
```

<class 'float'>

Na primeira linha do exemplo acima, calculamos a soma de 1.75 com 2 elevado à terceira potência. Em seguida, atribuímos o valor 1.25 à variável `taxa_juros` e a imprimimos usando `print(taxa_juros)`. Por fim, verificamos o tipo de dado da variável `taxa_juros` com `type(taxa_juros)`, que retorna `<class 'float'>`, indicando que é um número do tipo ponto flutuante.

5.2.3 O tipo de dado cadeia de caracteres

```
pais = "Brasil"  
  
print(pais)
```

Brasil

```
type(pais)
```

```
<class 'str'>
```

No código acima, criamos uma variável chamada `pais` e atribuímos a ela o valor “Brasil”, que é uma string. Em seguida, imprimimos o valor da variável `pais` usando `print(pais)`, o que exibe “Brasil” na tela. Por fim, verificamos o tipo de dado da variável `pais` com `type(pais)`, que retorna `<class 'str'>`, indicando que é uma string.

5.2.4 O tipo de dado lógico

O tipo de dados lógico, também conhecido como booleano (`bool`), é usado para representar valores de verdadeiro ou falso. No Python, os valores booleanos são `True` e `False`, que representam verdadeiro e falso, respectivamente.

```
1 == 2
```

False

```
5 % 2 == 0
```

False

```
taxa_juros_aumentando = True  
print(taxa_juros_aumentando)
```

True

```
type(taxa_juros_aumentando)
```

```
<class 'bool'>
```

Na primeira linha do exemplo acima, há uma verificação de igualdade entre 1 e 2, que retorna `False` porque 1 não é igual a 2. Em seguida, temos `5 % 2 == 0`, que também retorna `False` porque o resto da divisão de 5 por 2 não é igual a zero. Por fim, temos a variável `taxa_juros_aumentando` atribuída a `True`, indicando que a taxa de juros está aumentando. Ao imprimir e verificar o tipo dessa variável, obtemos `True` como resultado e o tipo `bool`, indicando que é um valor lógico.

5.2.5 Coerção de tipos

A coerção de dados em Python refere-se à conversão forçada de um tipo de dado para outro.

```
str_num = "1.41"  
type(str_num)
```

```
<class 'str'>
```

```
float(str_num)
```

```
1.41
```

Veja que a variável `str_num` é uma string que representa o número 1.41. Inicialmente, seu tipo é verificado usando a função `type`, que retorna `<class 'str'>`, indicando que é uma **string**. Em seguida, usamos a função `float()` para converter explicitamente `str_num` em um `float`. Após a conversão, o valor de `str_num` é “1.41” e seu tipo é alterado para `<class 'float'>`.

A seguir, outros exemplos de coerção.

Coerção para inteiro:

```
num_float = 3.14  
num_int = int(num_float)  
print(num_int)
```

```
3
```

Coerção para lógico:

```
valor_inteiro = 0  
valor_logico = bool(valor_inteiro)  
print(valor_logico)
```

```
False
```

Coerção para string (`str`):

```
num_float = 3.14
num_str = str(num_float)
print(num_str)
```

3.14

No código acima, estamos convertendo um valor float em string.

5.3 Objetos básicos

Em Python, há três estruturas de dados básicas: listas, tuplas e dicionários: - As listas são coleções ordenadas e mutáveis de elementos, permitindo a inclusão de itens de diferentes tipos e a modificação dos valores contidos nelas. - As tuplas são semelhantes às listas, porém são imutáveis, ou seja, não podem ser alteradas após a sua criação. - Já os dicionários são coleções não ordenadas de pares chave-valor, onde cada valor é associado a uma chave única, proporcionando acesso eficiente aos dados por meio das chaves.

5.3.1 Listas

Em Python, uma lista é uma estrutura de dados que permite armazenar uma coleção ordenada de elementos. Para criar uma lista, utilizamos colchetes [], e os elementos são separados por vírgulas. Podemos instanciar uma lista vazia simplesmente utilizando [] ou a função `list()`. Por exemplo:

```
lista_vazia = [] # Lista vazia
lista_vazia = list() # alternativa

# lista com PIB de países
pib_países = [1800, 2500, 3200, 5600, 6700]
```

No exemplo acima, também temos a lista `pib_países` que armazena o Produto Interno Bruto (PIB) de diferentes países. Para acessar elementos de uma lista em Python, podemos utilizar o índice do elemento desejado dentro de colchetes []. O índice começa do zero para o primeiro elemento, um para o segundo, e assim por diante. Por exemplo:

```
print(pib_países[1])
```

2500

Também podemos acessar os elementos a partir do final da lista utilizando índices negativos, onde -1 representa o último elemento, -2 o penúltimo, e assim por diante:

```
# Acessando o último elemento
ultimo_elemento = pib_países[-1]
print(ultimo_elemento)
```

6700

```
# Acessando o penúltimo elemento
penultimo_elemento = pib_países[-2]
print(penultimo_elemento)
```

5600

Podemos usar o método `append()` se desejamos adicionar um elemento ao final da lista, ou o método `insert()` se queremos adicionar um elemento em uma posição específica. Veja os exemplos de como usar ambos os métodos:

```
# Adicionando elementos ao final da lista usando append()
pib_países.append(2000) # Adiciona o valor 2000 ao final da lista

# Adicionando um elemento em uma posição específica usando insert()
pib_países.insert(1, 1500) # Adiciona o valor 1500 na posição 1 da lista
```

Para verificar o tamanho de uma lista em Python, podemos usar a função `len()`.

```
tamanho_lista = len(pib_países)
print("Tamanho da lista:", tamanho_lista)
```

Tamanho da lista: 7

Para ordenar uma lista, podemos usar o método `sort()` para ordenação *in-place* (ou seja, a lista é modificada) ou a função `sorted()` para retornar uma nova lista ordenada sem modificar a original:

```
# Usando a função sorted() para retornar uma nova lista ordenada
lista_ordenada = sorted(pib_países)
print("Nova lista ordenada:", lista_ordenada)
```

Nova lista ordenada: [1500, 1800, 2000, 2500, 3200, 5600, 6700]

```
# Ordenando a lista usando o método sort()
pib_paises.sort()
print("Lista ordenada:", pib_paises)
```

Lista ordenada: [1500, 1800, 2000, 2500, 3200, 5600, 6700]

O método `.pop()` é usado para remover e retornar o último elemento de uma lista. Também podemos especificar um índice para remover e retornar um elemento em uma posição específica da lista. Aqui está como usar o método `.pop()`:

```
# Removendo e retornando o último PIB da lista
ultimo_pib = pib_paises.pop()
print("Último PIB removido:", ultimo_pib)
```

Último PIB removido: 6700

```
print("Lista atualizada:", pib_paises)
```

Lista atualizada: [1500, 1800, 2000, 2500, 3200, 5600]

```
# Removendo e retornando o PIB de um país específico da lista
pib_removido = pib_paises.pop(1)
print("PIB removido:", pib_removido)
```

PIB removido: 1800

```
print("Lista atualizada:", pib_paises)
```

Lista atualizada: [1500, 2000, 2500, 3200, 5600]

5.3.2 Tuplas

As tuplas são estruturas de dados semelhantes às listas, mas com uma diferença fundamental: elas são imutáveis, ou seja, uma vez criadas, não podem ser modificadas. Elas são representadas por parênteses `()` em vez de colchetes `[]`.

```
x = (1, 2, 3) # tupla (lista imutável)
print(x)
```

(1, 2, 3)

```
# x[0] = 5 # gera erro!
```

Podemos usar tuplas para representar informações que não devem ser alteradas, como por exemplo, as taxas de câmbio entre moedas. Veja:

```
taxas_cambio = (("USD", "EUR", 0.82), ("USD", "JPY", 105.42), ("EUR", "JPY", 128.64))
```

Neste exemplo, temos uma tupla de tuplas que representam as taxas de câmbio entre o dólar (USD), o euro (EUR) e o iene japonês (JPY) para uma data fixa fictícia. Cada tupla interna contém três elementos: a moeda de origem, a moeda de destino e a taxa de câmbio. Como essas informações não devem ser alteradas, uma tupla é uma escolha apropriada.

Tip

Para acessar os elementos de uma tupla, você pode usar a mesma sintaxe que usa para acessar os elementos de uma lista, ou seja, usando colchetes [] e o índice do elemento desejado. Lembre-se de que os índices em Python começam em 0!

5.3.3 Dicionários

Dicionários em Python são estruturas de dados que permitem armazenar pares de chave-valor. Cada valor é associado a uma chave específica, permitindo o acesso rápido aos dados por meio das chaves, em vez de índices numéricos, como em listas e tuplas. Essa estrutura é útil quando você precisa associar informações de maneira semelhante a um banco de dados, onde você pode buscar informações com base em uma chave específica.

No exemplo abaixo, temos cotações de ações de algumas empresas brasileiras listadas na bolsa de valores.

```
cotacoes_acoes_brasileiras = {
    "PETR4": 36.75,
    "VALE3": 62.40,
    "ITUB4": 34.15,
    "BBDC4": 13.82
}
```


Cada chave é o código de negociação da ação na bolsa, e o valor associado é o preço da ação em uma data fixada. Por exemplo, `cotacoes_acoes_brasileiras["PETR4"]` retornaria o preço da ação da Petrobras.

Alternativamente, você pode criar um dicionário usando a função `dict`:

```
cotacoes_acoes_brasileiras = dict(PETR4=36.12, VALE3=62.40, ITUB4=34.15, BBDC4=13.82)
```

Você pode adicionar novos pares chave-valor a um dicionário ou atualizar os valores existentes. Por exemplo:

```
cotacoes_acoes_brasileiras["ABEV3"] = 12.80
cotacoes_acoes_brasileiras["PETR4"] = 36.75
```

Você pode remover pares chave-valor de um dicionário usando o comando método `pop()`. Por exemplo:

```
valor_removido = cotacoes_acoes_brasileiras.pop("BBDC4")
```

Outros métodos úteis para trabalhar com dicionários são `keys()`, `values()` e `items()` que retornam listas com as chaves, valores e itens do dicionário, respectivamente.

```
cotacoes_acoes_brasileiras.keys() # retorna uma lista contendo todas as chaves
```

```
dict_keys(['PETR4', 'VALE3', 'ITUB4', 'ABEV3'])
```

```
cotacoes_acoes_brasileiras.values() # retorna uma lista contendo todos os valores
```

```
dict_values([36.75, 62.4, 34.15, 12.8])
```

```
cotacoes_acoes_brasileiras.items() # retorna uma lista de tuplas
```

```
dict_items([('PETR4', 36.75), ('VALE3', 62.4), ('ITUB4', 34.15), ('ABEV3', 12.8)])
```

5.4 Fatias (*slices*)

O conceito de fatias, também conhecido como “*slicing*” em inglês, refere-se à técnica de extrair partes específicas de uma sequência, como uma lista, tupla ou string, usando índices. Ao utilizar fatias, você pode selecionar um intervalo de elementos dentro da sequência.

A sintaxe básica para fatias é `sequencia[inicio:fim:passo]`, onde:

- **inicio**: o índice inicial do intervalo a ser incluído na fatia (incluído).
- **fim**: o índice final do intervalo a ser incluído na fatia (excluído).
- **passo**: o tamanho do passo entre os elementos selecionados (opcional).

Considere a lista abaixo.

```
nomes_paises = ["Indonésia", "Índia", "Brasil", "África do Sul", "Alemanha"]
```

Para acessar os três primeiros países, podemos fazer:

```
nomes_paises[:3]
```

```
['Indonésia', 'Índia', 'Brasil']
```

Isso retorna os elementos da lista do índice 0 (inclusivo) ao índice 3 (exclusivo).

Se quisermos acessar os países do segundo ao terceiro:

```
nomes_paises[1:4]
```

```
['Índia', 'Brasil', 'África do Sul']
```

Podemos até mesmo fazer fatias reversas, onde o índice inicial é maior que o índice final, indicando que queremos percorrer a lista de trás para frente. Por exemplo, para acessar os últimos três países:

```
nomes_paises[-3:]
```

```
['Brasil', 'África do Sul', 'Alemanha']
```

Suponha que queremos acessar todos os países, mas pulando de dois em dois:

```
nomes_paises[::2]
```

```
['Indonésia', 'Brasil', 'Alemanha']
```

Neste exemplo, o `::2` indica que queremos começar do início da lista e ir até o final, pulando de dois em dois elementos.

5.5 Condicionais

O `if` e o `else` são estruturas de controle de fluxo em Python, usadas para tomar decisões com base em condições.

O bloco de código dentro do `if` é executado se a condição for avaliada como verdadeira (`True`). Por exemplo:

```
idade = 18
if idade >= 18:
    print("Você é maior de idade.")
```

Você é maior de idade.

O bloco de código dentro do `else` é executado se a condição do `if` for avaliada como falsa (`False`). Por exemplo:

```
idade = 16
if idade >= 18:
    print("Você é maior de idade.")
else:
    print("Você é menor de idade.")
```

Você é menor de idade.

Indentação

Em Python, a indentação é fundamental para definir blocos de código. No exemplo acima, observe que o código dentro do `if` e do `else` está indentado com quatro espaços. Isso indica que essas linhas pertencem ao bloco de código condicional. Se não houver indentação correta, o Python gerará um erro de sintaxe.

5.6 Estruturas repetitivas

As estruturas de repetição são utilizadas para executar um bloco de código repetidamente com base em uma condição específica. Existem duas principais estruturas de repetição em Python: `for` e `while`.

5.6.1 for

O loop `for` é utilizado para iterar sobre uma sequência (como uma lista, tupla, dicionário, etc.) e executar um bloco de código para cada item da sequência. Por exemplo:

```
for x in range(0, 20, 3): # lembre da notação dos slices
    print(x)
```

```
0
3
6
9
12
15
18
```

```
for pais in nomes_paises:
    print("País:", pais)
```

```
País: Indonésia
País: Índia
País: Brasil
País: África do Sul
País: Alemanha
```

No exemplo abaixo, temos uma lista de empresas e uma lista de lucros. Usando a função `zip()`, iteramos sobre essas duas listas em paralelo, imprimindo o nome da empresa e seu lucro correspondente. A função `zip()` combina elementos de duas ou mais sequências (como listas, tuplas, etc.) em pares ordenados.

```
empresas = ["Empresa A", "Empresa B", "Empresa C"]
lucros = [100000, 150000, 80000]

for empresa, lucro in zip(empresas, lucros):
    print("O lucro da empresa ", empresa, "foi R$", lucro)
```

```
O lucro da empresa  Empresa A foi R$ 100000
O lucro da empresa  Empresa B foi R$ 150000
O lucro da empresa  Empresa C foi R$ 80000
```

5.6.2 while

O `while` é uma estrutura de controle de fluxo que executa um bloco de código repetidamente enquanto uma condição especificada for verdadeira.

```
anos = 1
investimento = 1000
taxa_de_retorno = 0.05

while anos <= 10:
    investimento *= (1 + taxa_de_retorno)
    print("Após", anos, "anos, o investimento vale R$", round(investimento, 2))
    anos += 1
```

```
Após 1 anos, o investimento vale R$ 1050.0
Após 2 anos, o investimento vale R$ 1102.5
Após 3 anos, o investimento vale R$ 1157.62
Após 4 anos, o investimento vale R$ 1215.51
Após 5 anos, o investimento vale R$ 1276.28
Após 6 anos, o investimento vale R$ 1340.1
Após 7 anos, o investimento vale R$ 1407.1
Após 8 anos, o investimento vale R$ 1477.46
Após 9 anos, o investimento vale R$ 1551.33
Após 10 anos, o investimento vale R$ 1628.89
```

Neste exemplo, o loop calcula o valor do investimento ao longo de 10 anos, considerando um retorno anual de 5%. A cada iteração, o valor do investimento é atualizado multiplicando-se pelo fator de crescimento (`1 + taxa_de_retorno`).

5.7 Comprehensions

As comprehensions são uma maneira concisa e poderosa de criar coleções em Python, como listas, dicionários e conjuntos, a partir de iteráveis existentes, como listas, dicionários, conjuntos ou sequências. Elas permitem criar essas coleções de forma mais eficiente e legível em comparação com a abordagem tradicional de usar loops. As *comprehensions* podem incluir expressões condicionais para filtrar elementos ou expressões para transformar os elementos durante a criação da coleção.

Por exemplo, você pode criar uma lista de quadrados dos números de 1 a 10 usando uma compreensão de lista:

```
quadrados = [x ** 2 for x in range(1, 11)]
quadrados
```

[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

Isso é equivalente a:

```
quadrados = []
for x in range(1, 11):
    quadrados.append(x ** 2)
```

As *comprehensions* podem ser aplicadas a listas, dicionários e conjuntos, e você pode adicionar cláusulas condicionais para filtrar elementos com base em uma condição específica.

Abaixo, as variáveis `linhas` e `colunas` são definidas como intervalos de números de 1 a 3 e de 1 a 2, respectivamente. Em seguida, é utilizada uma compreensão de lista para gerar todos os pares possíveis, combinando cada valor de `linha` com cada valor de `coluna`. Por fim, um loop `for` é usado para iterar sobre a lista de pares e imprimir cada par na saída. O resultado será a impressão de todos os pares ordenados possíveis, combinando os valores de linha e coluna especificados.

```
linhas = range(1, 4)
colunas = range(1, 3)

pares = [(r, c) for r in linhas for c in colunas]

for x in pares:
    print(x)
```

(1, 1)
(1, 2)
(2, 1)
(2, 2)
(3, 1)
(3, 2)

Neste exemplo abaixo, a palavra “inconstitucionalissimamente” é analisada para contar quantas vezes cada letra aparece. Em seguida, é feito um loop sobre o dicionário resultante para imprimir a contagem de ocorrências de cada letra.

```

palavra = "inconstitucionalissimamente"

frequencia_letras = {letra: palavra.count(letra) for letra in palavra}

for letra, ocorrencias in frequencia_letras.items():
    print("A letra", letra, "ocorre", ocorrencias, "vezes" if ocorrencias > 1 else "vez")

```

```

A letra i ocorre 5 vezes
A letra n ocorre 4 vezes
A letra c ocorre 2 vezes
A letra o ocorre 2 vezes
A letra s ocorre 3 vezes
A letra t ocorre 3 vezes
A letra u ocorre 1 vez
A letra a ocorre 2 vezes
A letra l ocorre 1 vez
A letra m ocorre 2 vezes
A letra e ocorre 2 vezes

```

5.8 Funções

As funções são blocos de código reutilizáveis que realizam uma tarefa específica. Elas aceitam entradas, chamadas de argumentos, e podem retornar resultados.

Em Python, a sintaxe básica de uma função é a seguinte:

```

def nome_da_funcao(argumento1, argumento2, ...):
    # Corpo da função
    # Faça alguma coisa com os argumentos
    resultado = argumento1 + argumento2
    return resultado

```

Por exemplo, vamos criar uma função em Python chamada `calcular_juros` que calcula o montante final de um investimento com base no valor inicial, na taxa de juros e no número de anos:

```

def calcular_juros(valor_inicial, taxa_juros, anos):
    montante_final = valor_inicial * (1 + taxa_juros) ** anos
    return montante_final

```

Agora, definimos valores e chamamos a função:

```
investimento_inicial = 1000 # Valor inicial do investimento
taxa_juros_anual = 0.05     # Taxa de juros anual (5%)
anos = 5                    # Número de anos
resultado = calcular_juros(investimento_inicial, taxa_juros_anual, anos)
print("O montante final após", anos, "anos será de: R$", round(resultado, 2))
```

O montante final após 5 anos será de: R\$ 1276.28

5.8.1 Função lambda

Uma função lambda em Python é uma função anônima, o que significa que é uma função sem nome. Ela é definida usando a palavra-chave lambda e pode ter qualquer número de argumentos, mas apenas uma expressão. A sintaxe básica é a seguinte:

```
lambda argumento1, argumento2, ...: expressao
```

Veja um exemplo de uma função lambda que calcula o quadrado de um número:

```
quadrado = lambda x: x ** 2
```

Neste exemplo, `lambda x: x ** 2` cria uma função que aceita um argumento `x` e retorna `x` ao quadrado. Você pode então usar essa função da mesma forma que qualquer outra função. Por exemplo:

```
resultado = quadrado(5)
print(resultado)
```

25

As funções lambda são frequentemente usadas em situações em que você precisa de uma função temporária e simples, como em operações de mapeamento, filtragem e ordenação de dados.

5.9 Classes e objetos

Uma classe é uma estrutura que define o comportamento e as propriedades de um tipo de objeto. Podemos pensar em uma classe como uma representação de um conceito abstrato, como uma transação financeira ou um tipo específico de investimento.

Por exemplo, podemos criar uma classe chamada `Transacao` para representar uma transação financeira, com propriedades como o valor da transação, a data e o tipo de transação.

Um objeto, por outro lado, é uma instância específica de uma classe. Ele representa uma entidade concreta com suas próprias características e comportamentos. Continuando com o exemplo da classe `Transacao`, podemos criar objetos individuais para representar transações específicas, como a compra de ações de uma empresa em uma determinada data.

```
class Transacao:
    def __init__(self, valor, data, tipo):
        self.valor = valor
        self.data = data
        self.tipo = tipo

    def print_info(self):
        print(f"Tipo da transação: {self.tipo}, Valor: R${self.valor}, Data:{self.data}")
```

Neste exemplo, a classe `Transacao` possui um método especial `__init__` que é chamado quando um novo objeto é criado. Esse método inicializa as propriedades do objeto com os valores fornecidos como argumentos. Além disso, a classe possui um método chamado `print_info`, que imprime as informações da transação, incluindo o tipo, o valor e a data. Esse método também recebe `self` como parâmetro para acessar os atributos da instância atual da classe. Ao chamar `print_info()` em um objeto `Transacao`, ele exibirá as informações formatadas da transação.

A seguir, criamos dois objetos da classe `Transacao`, `transacao1` e `transacao2`, e acessamos suas propriedades para obter informações sobre as transações.

```
# Criando objetos da classe Transacao
transacao1 = Transacao(valor=1000, data="2024-03-11", tipo="Compra de ações")
transacao2 = Transacao(valor=500, data="2024-03-12", tipo="Venda de ações")

# Acessando as propriedades dos objetos
print("Valor da transação 1:", transacao1.valor)
```

Valor da transação 1: 1000

```
print("Data da transação 2:", transacao2.data)
```

Data da transação 2: 2024-03-12

```
# Acessando métodos dos objetos
transacao1.print_info()
```

Tipo da transação: Compra de ações, Valor: R\$1000, Data:2024-03-11

```
transacao2.print_info()
```

Tipo da transação: Venda de ações, Valor: R\$500, Data:2024-03-12

5.10 Exercícios

1. Trabalhando com tipos de dados básicos

a) Crie variáveis para representar dados econômicos, como o PIB de pelo menos três países, taxas de inflação ou taxas de desemprego. Use valores recentes e históricos para criar um conjunto diversificado de dados econômicos que representem diferentes contextos econômicos ao redor do mundo.

b) Realize operações matemáticas básicas com esses dados, como calcular médias, taxas de crescimento ou proporções. Por exemplo, você pode calcular médias dos valores do PIB, taxas de crescimento do PIB ao longo do tempo, proporções entre diferentes indicadores econômicos (como o PIB per capita em relação ao PIB total), entre outras operações.

2. Durante a análise de dados, pode ser necessário converter entre diferentes tipos de dados. Utilize os tipos de dados fundamentais (integers, floats, strings) e aplique coerção de tipos conforme necessário. Considere, por exemplo que as variáveis representando o PIB foram dadas em formato `string`. Então, converta um tipo `float` para computar a média dos PIBs considerados. Depois disso, converta novamente os valores de PIB em uma `string` para formatação de saída ser no seguinte formato 1,111%.

3. Utilize o código abaixo para gerar um pandas dataframe que representa o preço de fechamento de uma ação

```
import pandas as pd

# Criando uma lista de datas
datas = pd.date_range(start='2023-01-01', end='2023-12-31', freq='B') # Frequência 'B' pa

# Criando uma série de preços de fechamento simulados
import numpy as np
np.random.seed(0) # Define a semente aleatória para reprodutibilidade
```

```

precos_fechamento = np.random.normal(loc=50, scale=5, size=len(datas)) # Simulando preços

# Criando o DataFrame
df_precos_acoes = pd.DataFrame({'Data': datas, 'Preço de Fechamento': precos_fechamento})

# Exibindo as primeiras linhas do DataFrame
print(df_precos_acoes.head())

```

	Data	Preço de Fechamento
0	2023-01-02	58.820262
1	2023-01-03	52.000786
2	2023-01-04	54.893690
3	2023-01-05	61.204466
4	2023-01-06	59.337790

- Obtenha os preços de fechamento da ação durante do mês de janeiro determinado.
- Obtenha os preços de fechamento da ação entre 2023-06-01 e 2023-12-31.
- Obtenha os preços de fechamento da ação às sextas-feiras ao longo de todo o período. Calcule o preço de fechamento médio às segundas-feiras e compare com o das sextas-feiras. Qual deles apresenta maior desvio padrão?

```

# Criando uma coluna para armazenar o dia da semana
df_precos_acoes['Dia da Semana'] = df_precos_acoes['Data'].dt.weekday

# Exibindo as primeiras linhas do DataFrame com a nova coluna
print(df_precos_acoes.head())

```

	Data	Preço de Fechamento	Dia da Semana
0	2023-01-02	58.820262	0
1	2023-01-03	52.000786	1
2	2023-01-04	54.893690	2
3	2023-01-05	61.204466	3
4	2023-01-06	59.337790	4

- Considere a lista armazenada na variável `pib_anos` abaixo e utilize uma compreensão de lista para calcular o crescimento percentual do PIB em relação ao ano anterior para cada país.

```
# Lista de PIB de um país nos últimos cinco anos
pib_anos = [1000, 1200, 800, 1500, 2000] # Exemplo de valores fictícios para o PIB
```

5. Escreva uma função chamada `calcular_ipc` que receba três argumentos:

- `cesta_de_produtos`: Um dicionário que mapeia cada produto a sua quantidade na cesta.
- `precos_atual`: Um dicionário que mapeia cada produto ao seu preço atual.
- `precos_base`: Um dicionário que mapeia cada produto ao seu preço base (preço de referência). O IPC é calculado utilizando a seguinte fórmula:

$$IPC = \sum_i \frac{\text{preço atual do produto}_i}{\text{preço base do produto}_i} \times \text{quantidade do produto}_i.$$

A função deve retornar o valor calculado do IPC. Use o código abaixo para testar sua função.

```
# Cesta de produtos com suas respectivas quantidades
cesta_de_produtos = {'arroz': 1, 'feijao': 2, 'carne': 3}
# Preços atuais dos produtos
precos_atual = {'arroz': 5, 'feijao': 8, 'carne': 12}
# Preços base dos produtos
precos_base = {'arroz': 4, 'feijao': 7, 'carne': 10}

# Chamada da função para calcular o IPC
ipc = calcular_ipc(cesta_de_produtos, precos_atual, precos_base)
print("O Índice de Preços ao Consumidor (IPC) é:", ipc)
```

6. Você está encarregado de desenvolver um sistema para registrar e gerenciar transações de compra e venda de ações, além de calcular informações importantes sobre a carteira de investimentos. Para isso, você deve implementar duas classes em Python: `Transacao` e `Carteira`.

A classe `Transacao` representa uma única transação de compra ou venda de ações. Ela possui os seguintes atributos:

- **data**: uma string representando a data da transação no formato 'AAAA-MM-DD'. tipo: uma string indicando o tipo da transação, que pode ser 'compra' ou 'venda'.
- **valor**: um número inteiro representando a quantidade de ações transacionadas.

A classe `Carteira` representa a carteira de investimentos do usuário, que contém várias transações de ações. Ela possui os seguintes atributos: - **transacoes**: uma lista que armazena todas as transações de ações realizadas.

Além disso, a classe `Carteira` possui os seguintes métodos:

- **adicionar_transacao(transacao)**: adiciona uma nova transação à carteira.

- `calcular_posicao_atual(valor_atual_acao)`: calcula a posição atual da ação na carteira com base no valor atual da ação.
- `calcular_valor_presente(valor_atual_acao)`: calcula o valor presente da ação na carteira com base no valor atual da ação.

- Implemente as classes `Transacao` e `Carteira` com os atributos e métodos descritos acima.
- Crie três instâncias da classe `Transacao` para representar diferentes transações de compra e venda de ações.
- Crie uma instância da classe `Carteira` e adicione as transações criadas à carteira.
- Teste os métodos da classe `Carteira`, utilizando os exemplos fornecidos no código de teste abaixo.

```
# Criando algumas transações
transacao1 = Transacao('2024-03-18', 'compra', 10) # Compra de 10 ações
transacao2 = Transacao('2024-03-19', 'compra', 5)  # Compra de mais 5 ações
transacao3 = Transacao('2024-03-20', 'venda', 8)   # Venda de 8 ações

# Criando uma carteira e adicionando as transações
carteira = Carteira()
carteira.adicionar_transacao(transacao1)
carteira.adicionar_transacao(transacao2)
carteira.adicionar_transacao(transacao3)

# Valor atual da ação (hipotético)
valor_atual_acao = 50

# Testando os métodos da classe Carteira
posicao_atual = carteira.calcular_posicao_atual(valor_atual_acao)
valor_presente = carteira.calcular_valor_presente(valor_atual_acao)

# Exibindo os resultados
print("Posição atual da ação na carteira:", posicao_atual)
print("Valor presente na carteira:", valor_presente)
```

6 Processamento e visualização de dados

Notice!

Chapter to be translated!

6.1 Instalação de bibliotecas

A instalação de bibliotecas em Python é essencial para expandir a funcionalidade da linguagem. Existem várias maneiras de instalar bibliotecas, mas a mais comum é usando um gerenciador de pacotes. O pip é o gerenciador de pacotes padrão para Python e geralmente acompanha a instalação do Python.

Para instalar uma biblioteca com pip, abra o terminal ou prompt de comando e digite o seguinte comando:

```
pip install nome_da_biblioteca
```

Substitua `nome_da_biblioteca` pelo nome da biblioteca que você deseja instalar.

6.2 Processamento de dados numéricos

O NumPy (Numerical Python) é uma biblioteca essencial para computação numérica em Python. Ele fornece estruturas de dados eficientes para trabalhar com arrays multidimensionais e funções matemáticas poderosas para manipulação de dados.

Para instalar o NumPy, você pode usar o pip, que é o gerenciador de pacotes padrão do Python:

```
pip install numpy
```

O principal objeto em NumPy é o array multidimensional. Você pode criar arrays NumPy usando a função `numpy.array()` e realizar operações matemáticas básicas com eles:

```
import numpy as np

# Criando um array NumPy
arr = np.array([1, 2, 3, 4, 5])

# Operações matemáticas básicas
print("Soma:", np.sum(arr))
```

Soma: 15

```
print("Média:", np.mean(arr))
```

Média: 3.0

Além das operações básicas, o NumPy oferece funções universais (**ufuncs**) para aplicar operações em todos os elementos de um array de uma vez:

```
# Funções universais (ufuncs)
arr = np.array([1, 2, 3, 4, 5])
print("Quadrado de cada elemento:", np.square(arr))
```

Quadrado de cada elemento: [1 4 9 16 25]

```
print("Exponencial de cada elemento:", np.exp(arr))
```

Exponencial de cada elemento: [2.71828183 7.3890561 20.08553692 54.59815003 148.4131596]

A seguir, apresentamos três exemplos práticos de utilização das funções do NumPy.

Exemplo 1: Cálculo de Estatísticas Descritivas

O NumPy pode ser usado para calcular estatísticas descritivas, como média, mediana, desvio padrão, mínimo e máximo de séries temporais de dados econômicos, como o preço das ações de uma empresa ao longo do tempo.

```
import numpy as np

# Preço das ações de uma empresa ao longo do tempo (em dólares)
```

```
precos = np.array([100, 102, 105, 110, 108, 115, 120])

# Calculando estatísticas descritivas
print("Média:", np.mean(precos))
```

Média: 108.57142857142857

```
print("Desvio padrão:", np.std(precos))
```

Desvio padrão: 6.58693821908486

```
print("Máximo:", np.max(precos))
```

Máximo: 120

```
print("Mínimo:", np.min(precos))
```

Mínimo: 100

Exemplo 2: Análise de Séries Temporais O NumPy é útil para manipulação e análise de séries temporais. Por exemplo, você pode usar NumPy para calcular a taxa de retorno de um investimento ao longo do tempo ou para suavizar uma série temporal usando médias móveis.

```
precos = np.array([100, 102, 105, 110, 108, 115, 120])
# Calcular a taxa de retorno de um investimento ao longo do tempo
retornos = np.diff(precos) / precos[:-1] * 100
print("Taxa de retorno:", retornos)
```

Taxa de retorno: [2. 2.94117647 4.76190476 -1.81818182 6.48148148 4.34782609]

Tip

A função `np.diff` em NumPy é usada para calcular a diferença entre elementos consecutivos ao longo de um determinado eixo de um array. *Atenção:* O tamanho do retorno da função `np.diff` será sempre menor que o tamanho do vetor original de entrada por um

elemento. Por exemplo, se tivermos um vetor unidimensional com n elementos, a função `np.diff` retornará um vetor com $n - 1$ elementos, pois não há diferença para o último elemento.

```
precos = np.array([100, 102, 105, 110, 108, 115, 120])
# Suavizar uma série temporal usando médias móveis
tamanho_janela = 3
media_movel = np.convolve(precos, np.ones(tamanho_janela) / tamanho_janela, mode='valid')
print("Médias móveis:", media_movel)
```

Médias móveis: [102.33333333 105.66666667 107.66666667 111. 114.33333333]

Tip

A função `np.convolve` em NumPy é usada para realizar a convolução entre duas sequências, representadas por dois vetores unidimensionais. A convolução é uma operação matemática que combina duas funções para produzir uma terceira função que representa a quantidade de sobreposição entre elas conforme uma delas é deslocada ao longo do eixo. A sintaxe básica da função é `np.convolve(a, b, mode='full')`, onde `a` e `b` são os dois vetores unidimensionais a serem convolvidos e `mode` é um parâmetro opcional que define o modo de convolução. Os modos mais comuns são:

- `'full'`: Retorna a saída completa da convolução. O comprimento do resultado será `len(a) + len(b) - 1`.
- `'valid'`: Retorna apenas pontos onde as sequências se sobrepõem completamente. O comprimento do resultado será `max(len(a), len(b)) - min(len(a), len(b)) + 1`.
- `'same'`: Retorna a saída do mesmo tamanho que o vetor de entrada mais longo. O comprimento do resultado será `max(len(a), len(b))`.

No exemplo anterior, a função `np.convolve` foi usada aqui para calcular a média móvel dos preços. Nesse caso, a primeira sequência é o vetor de preços e a segunda sequência é um vetor de 1s dividido pelo tamanho da janela de média móvel. Isso cria uma sequência que representa uma média ponderada dos valores.

Exemplo 3: Simulação Monte Carlo O NumPy pode ser usado para realizar simulações Monte Carlo, que são amplamente utilizadas na modelagem financeira e na avaliação de risco. Por exemplo, você pode simular o desempenho de uma carteira de investimentos ao longo do tempo sob diferentes cenários de mercado.

```
# Simulação Monte Carlo do desempenho de uma carteira de investimentos
num_simulacoes = 1000
num_anos = 10
retorno_medio = 0.08
volatilidade = 0.15

# Gerar retornos aleatórios usando uma distribuição normal
retornos = np.random.normal(retorno_medio, volatilidade, size=(num_simulacoes, num_anos))

# Calcular o valor final da carteira para cada simulação
investimento_inicial = 10000
valores_finais = investimento_inicial * np.cumprod(1 + retornos, axis=1)

# Estatísticas descritivas dos valores finais da carteira
print("Valor final médio:", np.mean(valores_finais[:,-1]))
```

Valor final médio: 21614.1799688634

```
print("Desvio padrão dos valores finais:", np.std(valores_finais[:,-1]))
```

Desvio padrão dos valores finais: 9628.553110667357

Mais referências sobre NumPy:

- **Documentação oficial do NumPy:** <https://numpy.org/doc/stable/> A documentação oficial do NumPy contém informações detalhadas sobre todas as funções e métodos disponíveis, além de tutoriais e exemplos.
- **NumPy Quickstart Tutorial:** <https://numpy.org/doc/stable/user/quickstart.html> Este tutorial rápido fornece uma introdução rápida ao NumPy e suas funcionalidades básicas.

6.3 Análise e processamento de dados

O pandas é uma biblioteca de código aberto amplamente utilizada em Python para análise e manipulação de dados. Ele fornece estruturas de dados flexíveis e ferramentas poderosas para trabalhar com dados estruturados, facilitando a análise, limpeza e preparação de dados para diversas aplicações, como ciência de dados, finanças, pesquisa acadêmica e muito mais.

6.4 O que é pandas?

Pandas é uma biblioteca Python de código aberto que oferece estruturas de dados de alto desempenho e ferramentas de análise de dados. O pandas foi projetado para lidar com as complexidades do mundo real em análise de dados, oferecendo uma interface simples e intuitiva para trabalhar com dados tabulares.

Pandas é amplamente utilizado em análise de dados devido à sua capacidade de:

- Importar e exportar dados de uma variedade de fontes, incluindo arquivos CSV, Excel, SQL, JSON, HDF5 e mais.
- Manipular dados de forma eficiente, incluindo indexação, filtragem, agregação e limpeza.
- Realizar operações estatísticas e matemáticas em dados, como média, soma, desvio padrão, correlação, etc.

As duas principais estruturas de dados fornecidas pelo pandas são series e dataframes.

6.4.1 Séries

Uma **Série** é uma estrutura de dados unidimensional que pode conter qualquer tipo de dados, como inteiros, floats, strings, entre outros. Cada elemento em uma Série possui um rótulo único chamado de índice. A Série é semelhante a uma lista ou array unidimensional em Python, mas fornece recursos adicionais, como operações vetorizadas e alinhamento automático de dados com base nos rótulos de índice.

Suponha que temos uma Série representando os preços diários de uma ação:

Data	Preço
2024-03-18	100
2024-03-19	105
2024-03-20	98
2024-03-21	102

Podemos criar uma Serie pandas para representar esses dados:

```
import pandas as pd

# Dados dos preços da ação
data = ['2024-03-18', '2024-03-19', '2024-03-20', '2024-03-21']
precos = [100, 105, 98, 102]

# Criando uma Série pandas
```

```
serie_precos_acao = pd.Series(precos, index=pd.to_datetime(data), name='Preço da Ação')
print(serie_precos_acao)
```

```
2024-03-18    100
2024-03-19    105
2024-03-20     98
2024-03-21    102
Name: Preço da Ação, dtype: int64
```

6.4.2 DataFrame

DataFrame é uma estrutura de dados bidimensional semelhante a uma tabela de banco de dados ou uma planilha do Excel. Ele é composto por linhas e colunas, onde cada coluna pode conter um tipo de dado diferente. Cada coluna e linha em um DataFrame possui um rótulo exclusivo chamado de índice e nome, respectivamente. O DataFrame permite realizar uma ampla gama de operações de manipulação e análise de dados, como indexação, filtragem, agregação, limpeza, entre outras.

Suponha que temos um DataFrame representando os preços diários de várias ações. Podemos criar um DataFrame pandas para representar esses dados. Veja no exemplo abaixo.

```
import pandas as pd
import numpy as np

# Dados dos preços das ações
data = ['2024-03-18', '2024-03-19', '2024-03-20', '2024-03-21']
precos_acoes = {
    'Ação 1': [100, 105, np.nan, 102],
    'Ação 2': [50, 52, 48, 49],
    'Ação 3': [75, np.nan, 72, 74]
}

# Criando um DataFrame pandas
df_precos_acoes = pd.DataFrame(precos_acoes, index=pd.to_datetime(data))
print(df_precos_acoes)
```

	Ação 1	Ação 2	Ação 3
2024-03-18	100.0	50	75.0
2024-03-19	105.0	52	NaN
2024-03-20	NaN	48	72.0
2024-03-21	102.0	49	74.0

6.4.3 Principais funcionalidades

A função `df.isna()` é uma função fornecida pelo pandas em um `DataFrame` (`df`) que retorna uma matriz booleana indicando se cada elemento do `DataFrame` é um valor ausente (`NaN`).

Quando aplicada a um `DataFrame`, a função `isna()` retorna um `DataFrame` com o mesmo formato, onde cada valor é substituído por `True` se for `NaN` e `False` caso contrário.

Isso é útil para identificar rapidamente os valores ausentes em um `DataFrame` e realizar operações de limpeza ou tratamento de dados, como preenchimento de valores ausentes ou remoção de linhas ou colunas contendo esses valores.

Se aplicarmos `df_precos_acoes.isna()`, obteremos:

```
df_precos_acoes.isna()
```

	Ação 1	Ação 2	Ação 3
2024-03-18	False	False	False
2024-03-19	False	False	True
2024-03-20	True	False	False
2024-03-21	False	False	False

Para contar a quantidade de `NaN` em cada coluna, combine `is.na()` com `sum()`:

```
df_precos_acoes.isna().sum()
```

```
Ação 1    1
Ação 2    0
Ação 3    1
dtype: int64
```

O método `dropna()` no pandas é usado para remover linhas ou colunas que contenham valores ausentes (`NaN`).

```
df_precos_acoes.dropna()
```

	Ação 1	Ação 2	Ação 3
2024-03-18	100.0	50	75.0
2024-03-21	102.0	49	74.0

O parâmetro `subset` é usado para especificar em quais colunas ou linhas o pandas deve procurar por valores ausentes antes de remover. Quando usamos `df.dropna(subset=["Ação 3"])`, estamos instruindo o pandas a remover todas as linhas onde houver um valor ausente na coluna “Ação 3”.

```
df_precos_acoes.dropna(subset=["Ação 3"])
```

	Ação 1	Ação 2	Ação 3
2024-03-18	100.0	50	75.0
2024-03-20	NaN	48	72.0
2024-03-21	102.0	49	74.0

Na função `dropna()`, o parâmetro `inplace=True` especifica que a modificação deve ser feita diretamente no DataFrame original, em vez de retornar um novo DataFrame sem os valores ausentes. Quando `inplace=True` é usado com `dropna()`, o DataFrame original é modificado e as linhas ou colunas com valores ausentes são removidas permanentemente.

```
df_precos_acoes.dropna(inplace = True)
```

A função `fillna()` no pandas é usada para preencher valores ausentes (NaN) em um DataFrame com um valor específico.

Considere o seguinte DataFrame `df` que representa os dados de clientes de um banco com alguns dados faltantes:

```
import pandas as pd
import numpy as np

dados = {'Nome': ['João', 'Maria', 'Pedro', 'Ana', 'Mariana'],
        'Idade': [25, 30, np.nan, 40, 35],
        'Renda Mensal': [5000, 6000, np.nan, 4500, 5500],
        'Limite de Crédito': [10000, np.nan, 8000, 12000, np.nan]}
df_clientes = pd.DataFrame(dados)
```

Neste exemplo,

- Os valores ausentes na coluna “Idade” foram preenchidos com a mediana das idades existentes no DataFrame.
- Os valores ausentes na coluna “Limite de Crédito” foram preenchidos com a moda dos limites de crédito existentes no DataFrame.
- Os valores ausentes na coluna “Renda Mensal” foram preenchidos com a média das rendas mensais existentes no DataFrame.

```
# Preenchendo valores ausentes na coluna 'Idade' com a mediana das idades
df_clientes['Idade'] = df_clientes['Idade'].fillna(df_clientes['Idade'].median())

# Preenchendo valores ausentes na coluna 'Limite de Crédito' com a moda dos limites de crédito
df_clientes['Limite de Crédito'] = df_clientes['Limite de Crédito'].fillna(df_clientes['Limite de Crédito'].mode()[0])

# Preenchendo valores ausentes na coluna 'Renda Mensal' com a média das rendas mensais
df_clientes['Renda Mensal'] = df_clientes['Renda Mensal'].fillna(df_clientes['Renda Mensal'].mean())

df_clientes
```

	Nome	Idade	Renda Mensal	Limite de Crédito
0	João	25.0	5000.0	10000.0
1	Maria	30.0	6000.0	8000.0
2	Pedro	32.5	5250.0	8000.0
3	Ana	40.0	4500.0	12000.0
4	Mariana	35.0	5500.0	8000.0

Agora, vamos carregar os dados `gapminder`, que está no arquivo `gapminder.zip`.

```
gapminder = pd.read_csv("data/gapminder.zip", sep = "\t")
```

A função `head()` é usada para visualizar as primeiras linhas do conjunto de dados `gapminder`, oferecendo uma rápida visão geral da sua estrutura e conteúdo.

```
gapminder.head()
```

	country	continent	year	lifeExp	pop	gdpPercap
0	Afghanistan	Asia	1952	28.801	8425333	779.445314
1	Afghanistan	Asia	1957	30.332	9240934	820.853030
2	Afghanistan	Asia	1962	31.997	10267083	853.100710
3	Afghanistan	Asia	1967	34.020	11537966	836.197138
4	Afghanistan	Asia	1972	36.088	13079460	739.981106

O método `info()` fornece informações sobre o conjunto de dados, incluindo o número de entradas, o tipo de dados de cada coluna e se há valores nulos.

```
gapminder.info()
```

```

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1704 entries, 0 to 1703
Data columns (total 6 columns):
#   Column      Non-Null Count  Dtype
---  -
0   country     1704 non-null   object
1   continent    1704 non-null   object
2   year         1704 non-null   int64
3   lifeExp      1704 non-null   float64
4   pop          1704 non-null   int64
5   gdpPercap    1704 non-null   float64
dtypes: float64(2), int64(2), object(2)
memory usage: 80.0+ KB

```

A função `describe()` gera estatísticas descritivas para cada coluna numérica do conjunto de dados, como contagem, média, desvio padrão, mínimo e máximo.

```
gapminder.describe()
```

	year	lifeExp	pop	gdpPercap
count	1704.000000	1704.000000	1.704000e+03	1704.000000
mean	1979.500000	59.474439	2.960121e+07	7215.327081
std	17.26533	12.917107	1.061579e+08	9857.454543
min	1952.000000	23.599000	6.001100e+04	241.165876
25%	1965.750000	48.198000	2.793664e+06	1202.060309
50%	1979.500000	60.712500	7.023596e+06	3531.846988
75%	1993.250000	70.845500	1.958522e+07	9325.462346
max	2007.000000	82.603000	1.318683e+09	113523.132900

A função `value_counts()` conta o número de ocorrências de cada categoria na coluna “continent” do conjunto de dados `gapminder`, transforma os resultados em um `DataFrame`, renomeia as colunas para “continent” e “n” (indicando a contagem), e reconfigura o índice.

```
gapminder.value_counts("continent").to_frame("n").reset_index()
```

	continent	n
0	Africa	624
1	Asia	396
2	Europe	360
3	Americas	300
4	Oceania	24

No trecho abaixo, fazemos uma contagem de valores para as combinações únicas de categorias nas colunas “continent” e “year” do DataFrame gapminder. Os resultados são transformados em um DataFrame, renomeados como “continent”, “year” e “n” (indicando a contagem), e o índice é reconfigurado.

```
gapminder.value_counts(["continent", "year"]).to_frame("n").reset_index()
```

	continent	year	n
0	Africa	1952	52
1	Africa	1987	52
2	Africa	1957	52
3	Africa	2002	52
4	Africa	1997	52
5	Africa	1992	52
6	Africa	2007	52
7	Africa	1982	52
8	Africa	1977	52
9	Africa	1972	52
10	Africa	1967	52
11	Africa	1962	52
12	Asia	1952	33
13	Asia	2007	33
14	Asia	2002	33
15	Asia	1997	33
16	Asia	1992	33
17	Asia	1987	33
18	Asia	1977	33
19	Asia	1972	33
20	Asia	1967	33
21	Asia	1962	33
22	Asia	1957	33
23	Asia	1982	33
24	Europe	1982	30
25	Europe	1957	30
26	Europe	2007	30
27	Europe	2002	30
28	Europe	1997	30
29	Europe	1992	30
30	Europe	1987	30
31	Europe	1977	30
32	Europe	1972	30
33	Europe	1967	30

```

34   Europe  1962  30
35   Europe  1952  30
36  Americas  2002  25
37  Americas  2007  25
38  Americas  1952  25
39  Americas  1962  25
40  Americas  1967  25
41  Americas  1972  25
42  Americas  1977  25
43  Americas  1982  25
44  Americas  1987  25
45  Americas  1992  25
46  Americas  1997  25
47  Americas  1957  25
48  Oceania  1982   2
49  Oceania  2002   2
50  Oceania  1997   2
51  Oceania  1992   2
52  Oceania  1987   2
53  Oceania  1957   2
54  Oceania  1977   2
55  Oceania  1972   2
56  Oceania  1967   2
57  Oceania  1962   2
58  Oceania  1952   2
59  Oceania  2007   2

```

6.5 Dados organizados (tidy data)

Todas as tabelas abaixo tem o mesmo dado (foram tiradas do pacote `tidyr` do R), que mostra a quantidade de casos de uma doença e a população total de alguns países.

```

table1 = pd.read_csv("data/table1.csv")
table2 = pd.read_csv("data/table2.csv")
table3 = pd.read_csv("data/table3.csv")
table4a = pd.read_csv("data/table4a.csv")
table4b = pd.read_csv("data/table4b.csv")

```

```
table1
```

```
country year  cases  population
```

```

0 Afghanistan 1999      745    19987071
1 Afghanistan 2000     2666    20595360
2      Brazil 1999    37737    172006362
3      Brazil 2000    80488    174504898
4        China 1999  212258   1272915272
5        China 2000  213766   1280428583

```

table2

```

      country year      type      count
0  Afghanistan 1999      cases        745
1  Afghanistan 1999 population   19987071
2  Afghanistan 2000      cases        2666
3  Afghanistan 2000 population   20595360
4      Brazil 1999      cases       37737
5      Brazil 1999 population   172006362
6      Brazil 2000      cases       80488
7      Brazil 2000 population   174504898
8        China 1999      cases       21258
9        China 1999 population  1272915272
10       China 2000      cases       213766
11       China 2000 population  1280428583

```

table3

```

      country year      rate
0  Afghanistan 1999    745/19987071
1  Afghanistan 2000   2666/20595360
2      Brazil 1999   37737/172006362
3      Brazil 2000   80488/174504898
4        China 1999  212258/1272915272
5        China 2000  213766/1280428583

```

table4a

```

      country  1999  2000
0  Afghanistan   745   2666
1      Brazil  37737  80488
2        China 212258 213766

```

```
table4b
```

	country	1999	2000
0	Afghanistan	19987071	20595360
1	Brazil	172006362	174504898
2	China	1272915272	1280428583

O exemplo abaixo cria uma nova coluna chamada `rate` no DataFrame `table1`. A função `assign` adiciona uma nova coluna ao DataFrame, enquanto a expressão `lambda` calcula os valores para essa nova coluna.

```
table1.assign(rate = lambda _: 10000 * (_.cases / _.population))
```

	country	year	cases	population	rate
0	Afghanistan	1999	745	19987071	0.372741
1	Afghanistan	2000	2666	20595360	1.294466
2	Brazil	1999	37737	172006362	2.193930
3	Brazil	2000	80488	174504898	4.612363
4	China	1999	212258	1272915272	1.667495
5	China	2000	213766	1280428583	1.669488

No exemplo abaixo, agrupamos os dados do DataFrame `table1` pela coluna “year” (ano) e depois calcula a soma dos casos para cada ano. O método `groupby("year")` agrupa os dados por ano, criando grupos separados para cada ano. `as_index = False` especifica que a coluna usada para agrupamento (“year”) não deve ser definida como índice no DataFrame resultante. O método `agg` é usado para realizar uma operação de agregação nos grupos. Aqui, `np.sum` é usado para calcular a soma dos valores da coluna “cases” para cada grupo.

```
(table1.groupby("year", as_index = False)
     .agg(total_cases = ("cases", np.sum)))
```

	year	total_cases
0	1999	250740
1	2000	296920

Para fazer o mesmo com os dados da `table1`, temos que usar a função `pivot_table`:

```
table2_tidy = (table2.pivot_table(index = ["country", "year"], columns = "type", values =
                                   .reset_index())
```

```

        .rename_axis(None, axis = 1))

table2_tidy.assign(rate = lambda _: 10000 * (_.cases / _.population))

```

	country	year	cases	population	rate
0	Afghanistan	1999	745.0	1.998707e+07	0.372741
1	Afghanistan	2000	2666.0	2.059536e+07	1.294466
2	Brazil	1999	37737.0	1.720064e+08	2.193930
3	Brazil	2000	80488.0	1.745049e+08	4.612363
4	China	1999	212258.0	1.272915e+09	1.667495
5	China	2000	213766.0	1.280429e+09	1.669488

No exemplo acima, usamos o método `pivot_table` do pandas para reorganizar os dados do DataFrame `table2`. Ele reorganiza os dados de forma que os valores da coluna “count” sejam pivotados (transformados em colunas) com base nos valores únicos da combinação de “country” e “year”. Os parâmetros `index`, `columns` e `values` especificam respectivamente as colunas que serão usadas como índice, as que serão transformadas em colunas e os valores a serem preenchidos na tabela pivô. Após a operação de pivotagem, são encadeados métodos adicionais para modificar a estrutura do DataFrame resultante:

- `reset_index()` redefine os índices do DataFrame para índices numéricos padrão, movendo os índices anteriores (no caso, “country” e “year”) para colunas.
- `rename_axis(None, axis=1)` remove os nomes dos índices das colunas, substituindo-os por `None`. Isso é feito especificamente para limpar os nomes das colunas do DataFrame.

Após a transformação dos dados, a função `assign` é usada para criar uma nova coluna chamada `rate` no DataFrame resultante `table2_tidy`.

Agora, vamos fazer o mesmo para a `table4a` e `table4b`:

```

table4_tidy = (table4a.melt(id_vars = "country", value_vars = ["1999", "2000"], var_name =
                    .merge(table4b.melt(id_vars = "country", value_vars = ["1999", "2000"], va
                        on = ("country", "year")))

table4_tidy.assign(rate = lambda _: 10000 * (_.cases / _.population))

```

	country	year	cases	population	rate
0	Afghanistan	1999	745	19987071	0.372741
1	Brazil	1999	37737	172006362	2.193930
2	China	1999	212258	1272915272	1.667495
3	Afghanistan	2000	2666	20595360	1.294466

4	Brazil	2000	80488	174504898	4.612363
5	China	2000	213766	1280428583	1.669488

Os DataFrames `table4a` e `table4b` são derretidos usando o método `melt`:

- Para `table4a`, as colunas que permanecerão fixas são especificadas através do argumento `id_vars = "country"`, enquanto as colunas "1999" e "2000" são derretidas como variáveis usando `value_vars = ["1999", "2000"]`. Os nomes das variáveis derretidas são renomeadas para "year" e "cases" usando `var_name = "year"` e `value_name = "cases"`, respectivamente.
- Da mesma forma, para `table4b`, as colunas "country" e "1999", "2000" são derretidas, com os nomes das variáveis renomeadas para "year" e "population", respectivamente.

Os DataFrames resultantes do derretimento de `table4a` e `table4b` são mesclados usando o método `merge`. A mesclagem é feita com base nas colunas "country" e "year", garantindo que os dados correspondentes de `table4a` e `table4b` sejam combinados corretamente.

Finalmente, o método `assign` é usado para criar uma nova coluna chamada "rate", que representa a taxa de casos por 10.000 habitantes.

Para a `table3`, basta separar a coluna `cases` considerando o separador `\`:

```
print(table3)
```

	country	year	rate
0	Afghanistan	1999	745/19987071
1	Afghanistan	2000	2666/20595360
2	Brazil	1999	37737/172006362
3	Brazil	2000	80488/174504898
4	China	1999	212258/1272915272
5	China	2000	213766/1280428583

```
table3_tidy = (table3.assign(cases = lambda _: _.rate.str.split("/", expand = True)[0].astype(int),
                             population = lambda _: _.rate.str.split("/", expand = True)[1].astype(int))
                 .drop("rate", axis = 1))
```

```
table3_tidy
```

	country	year	cases	population
0	Afghanistan	1999	745	19987071
1	Afghanistan	2000	2666	20595360

2	Brazil	1999	37737	172006362
3	Brazil	2000	80488	174504898
4	China	1999	212258	1272915272
5	China	2000	213766	1280428583

```
table3_tidy.assign(rate = lambda _: 10000 * (_.cases / _.population))
```

	country	year	cases	population	rate
0	Afghanistan	1999	745	19987071	0.372741
1	Afghanistan	2000	2666	20595360	1.294466
2	Brazil	1999	37737	172006362	2.193930
3	Brazil	2000	80488	174504898	4.612363
4	China	1999	212258	1272915272	1.667495
5	China	2000	213766	1280428583	1.669488

O parâmetro `expand` é utilizado no método `str.split()` para especificar se o resultado da divisão deve ser expandido em um `DataFrame` (se `True`) ou mantido como uma lista de valores (se `False`, que é o padrão).

Principais funcionalidades - ver aulas paulo

6.6 Visualização de dados

Duas bibliotecas amplamente utilizadas para visualização em Python são o `Matplotlib` e o `Plotnine`. O `Matplotlib` oferece uma ampla gama de opções para criar visualizações estáticas, desde gráficos simples até gráficos complexos e personalizados. Por outro lado, o `Plotnine` é uma biblioteca baseada na gramática de gráficos (parecido com o `ggplot2` do R), o que facilita a criação de visualizações elegantes e concisas usando uma sintaxe intuitiva e expressiva.

6.6.1 Matplotlib

Antes de começarmos a criar visualizações, é importante entender alguns conceitos básicos do `Matplotlib`:

- **Figura e Eixo (Axes):** No `Matplotlib`, uma figura é a janela ou página na qual tudo é desenhado. Dentro de uma figura, pode haver vários eixos (ou subplots), onde os dados são efetivamente plotados.

- **Método plot():** O método plot() é usado para criar gráficos de linha, pontos ou marcadores. Ele aceita uma variedade de argumentos para personalizar a aparência do gráfico, como cor, estilo de linha, largura da linha, etc.
- **Customização:** O Matplotlib oferece muitas opções de personalização para ajustar a aparência dos gráficos, incluindo a adição de rótulos aos eixos, título do gráfico, legendas, entre outros.

Agora, vamos ver um exemplo de como criar um gráfico de pontos usando dados fictícios, onde cada unidade de dado está relacionada a uma empresa.

```
import matplotlib.pyplot as plt

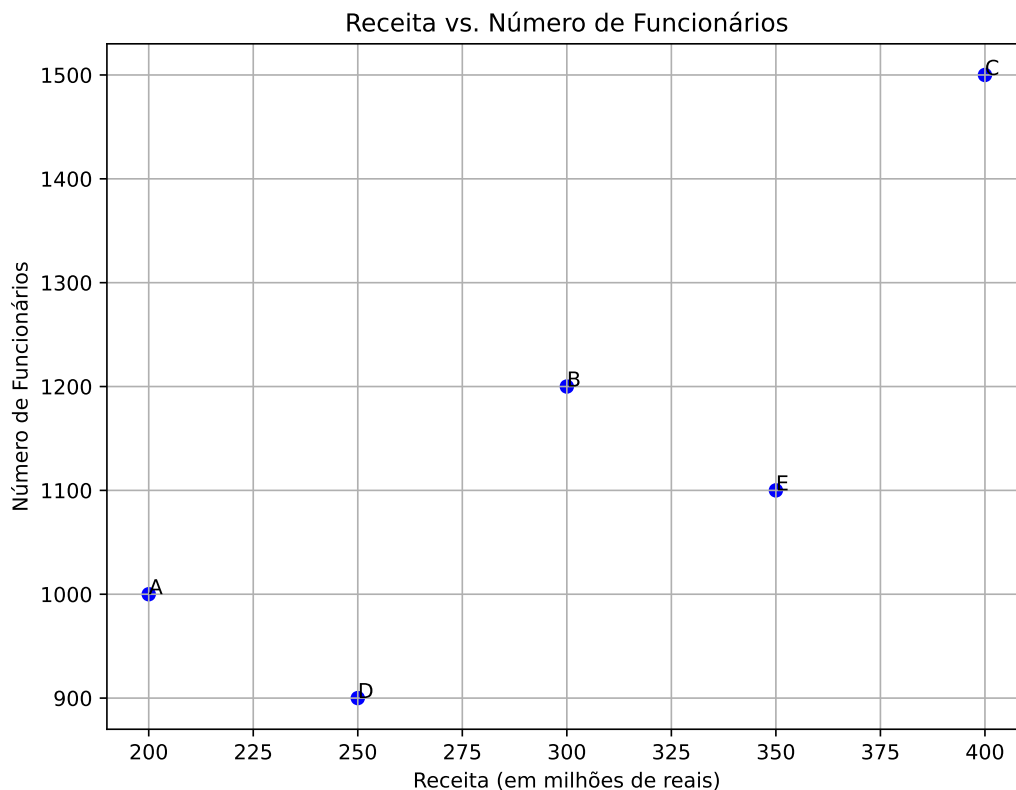
# Dados de exemplo: Nome das empresas, receita e número de funcionários
empresas = ['A', 'B', 'C', 'D', 'E']
receita = [200, 300, 400, 250, 350] # em milhões de reais
funcionarios = [1000, 1200, 1500, 900, 1100]

# Criando o gráfico de pontos
plt.figure(figsize=(8, 6))
plt.scatter(receita, funcionarios, color='blue', marker='o')

# Adicionando rótulos e título
plt.xlabel('Receita (em milhões de reais)')
plt.ylabel('Número de Funcionários')
plt.title('Receita vs. Número de Funcionários')

# Adicionando anotações para cada ponto
for i in range(len(empresas)):
    plt.annotate(empresas[i], (receita[i], funcionarios[i]))

# Exibindo o gráfico
plt.grid(True)
plt.show()
```

Neste exemplo, cada ponto no gráfico representa uma empresa, onde o eixo x representa a receita (em milhões de reais) e o eixo y representa o número de funcionários. As anotações são usadas para identificar cada empresa no gráfico.

Na sequência, utilizamos o Matplotlib para criar um gráfico de linha que representa a evolução das vendas de dois produtos ao longo de vários anos. Cada ponto no gráfico representa o número de vendas em um ano específico.

```
import matplotlib.pyplot as plt

# Dados de exemplo: Anos e vendas de produtos
anos = [2010, 2011, 2012, 2013, 2014, 2015, 2016]
vendas_produto_A = [500, 600, 550, 700, 800, 750, 900]
vendas_produto_B = [400, 450, 500, 550, 600, 650, 700]

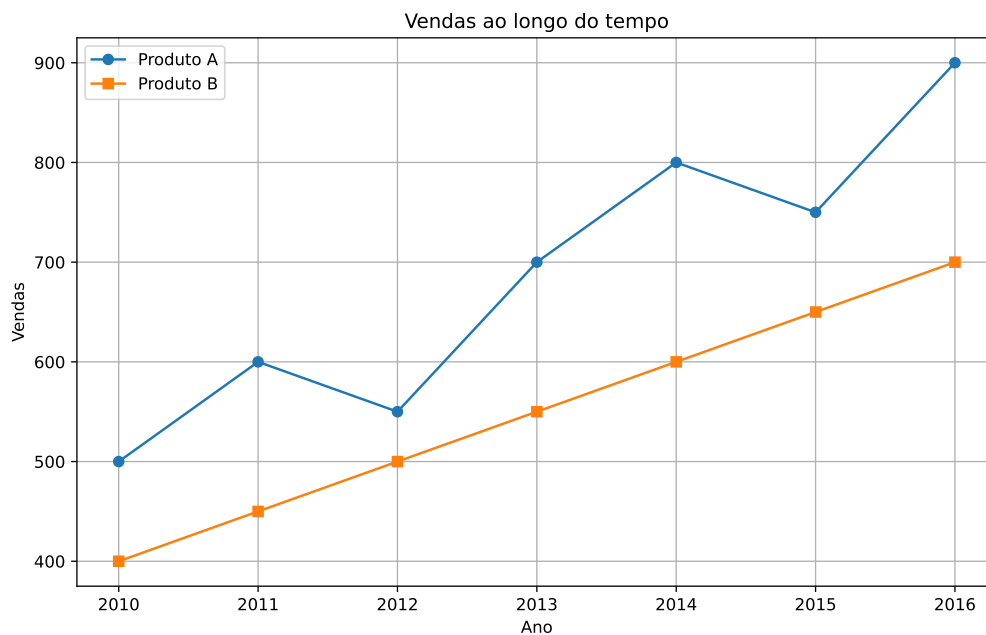
# Criando o gráfico de linha
```

```
plt.figure(figsize=(10, 6))
plt.plot(anos, vendas_produto_A, marker='o', label='Produto A')
plt.plot(anos, vendas_produto_B, marker='s', label='Produto B')

# Adicionando rótulos e título
plt.xlabel('Ano')
plt.ylabel('Vendas')
plt.title('Vendas ao longo do tempo')

# Adicionando legenda
plt.legend()

# Exibindo o gráfico
plt.grid(True)
plt.show()
```



6.6.2 Plotnine

Plotnine é uma biblioteca em Python que permite criar visualizações de dados estatísticos de uma forma simples e concisa, utilizando a gramática de gráficos do R (também conhecida

como ggplot2). Essa gramática consiste em uma abordagem declarativa para a construção de gráficos, onde os elementos visuais são adicionados em camadas para formar o gráfico final.

```
from plotnine import *
```

Para exemplificar, vamos utilizar a base de dados `gapminder`.

```
((ggplot(gapminder, aes(x = "continent", fill = "continent"))) +  
  geom_bar(aes(y = "stat(count) / 12"), alpha = 0.75) +  
  labs(x = "", y = "Number of countries", title = "Continents") +  
  theme(legend_position = "none") +  
  coord_flip()+  
  theme_bw())  
.show())
```

```
((ggplot(gapminder, aes(x = "lifeExp", y = "stat(density)"))) +  
  geom_histogram(fill = "blue", color = "white", alpha = 0.5) +  
  labs(x = "Life Expectancy", y = "", title = "Gapminder"))  
.show())
```

```
((ggplot(gapminder, aes(x = "lifeExp", y = "stat(density)"))) +  
  geom_histogram(fill = "blue", color = "white", alpha = 0.5) +  
  labs(x = "Life Expectancy", y = "", title = "Gapminder") +  
  facet_wrap("~ continent", nrow = 1) +  
  theme(figure_size = (12, 2))).  
show)
```

```
((gapminder.groupby(["continent", "year"], as_index = False)  
  .agg(median_lifeExp = ("lifeExp", np.median))  
  .pipe(lambda _: ggplot(_, aes(x = "year", y = "median_lifeExp", color = "continent"))  
    geom_line(size = 0.75) +  
    geom_point(size = 1.5) +  
    labs(x = "Year", y = "Median Life Expectancy", color = "Continen")  
    .show())
```

Part III

Case studies

References

- McKinney, Wes. 2022. *Python for Data Analysis*. " O'Reilly Media, Inc."
- Rosling, Hans. 2012. "Data - Gapminder.org" <http://www.gapminder.org/data/>.
- Wilkinson, Leland. 2012. *The Grammar of Graphics*. Springer.