

# The ROCK book

Szilvia Zörgő & Gjalt-Jorn Peters

2022-09-15 at 17:09:11 UTC (UTC+0000)



# Contents

	7
<b>I Qualitative Research</b>	<b>9</b>
1 Qualitative research	11
2 Quantitative Ethnography	13
3 Qualitative methods in psychology	15
4 Cognitive interviews	17
4.1 Cognitive validity, measures, and manipulations . . . . .	17
4.2 Response processes . . . . .	18
4.3 Cognitive interviews . . . . .	19
4.4 Common cognitive interview coding schemes . . . . .	20
4.5 Coding cognitive interviews . . . . .	20
4.6 Towards test validity: response models . . . . .	21
4.7 Response processes revisited . . . . .	22
4.8 Selecting response model parts . . . . .	22
4.9 The response process spectrum . . . . .	22
4.10 Coding schemes based on response spectra . . . . .	23
4.11 Preparing prompts . . . . .	23
<b>5 Initial codes</b>	<b>25</b>
5.1 Code organization . . . . .	25
5.2 Compiling your initial set of codes . . . . .	27
5.3 Developing coding instructions . . . . .	29
5.4 Segmentation . . . . .	30
5.5 Your initial code book . . . . .	32

<b>6</b>	<b>Planning and Conducting Interviews</b>	<b>35</b>
6.1	Planning . . . . .	35
<b>7</b>	<b>Coding</b>	<b>37</b>
7.1	Coding . . . . .	37
7.2	Segmenting . . . . .	37
<b>8</b>	<b>Preregistering qualitative research</b>	<b>39</b>
8.1	Types of registration . . . . .	39
8.2	Common registration form items . . . . .	40
<b>9</b>	<b>Reporting Qualitative Research</b>	<b>43</b>
9.1	Introduction . . . . .	43
9.2	Methods . . . . .	43
9.3	Results . . . . .	44
9.4	Discussion . . . . .	44
9.5	Reporting standards . . . . .	44
<b>II</b>	<b>The ROCK</b>	<b>47</b>
<b>10</b>	<b>The ROCK vocabulary</b>	<b>49</b>
10.1	ROCK terms . . . . .	49
10.2	ENA terms . . . . .	52
<b>11</b>	<b>The ROCK standard</b>	<b>53</b>
11.1	Examples . . . . .	56
<b>12</b>	<b>The rock R package</b>	<b>59</b>
12.1	Downloading and installing R and RStudio . . . . .	59
12.2	Downloading and installing the rock package . . . . .	60
12.3	rock functions . . . . .	61
12.4	rock options and defaults . . . . .	62
<b>13</b>	<b>Recoding with the rock R package</b>	<b>63</b>
13.1	Initial coding versus recoding . . . . .	63
13.2	Deleting codes . . . . .	63
<b>14</b>	<b>The iROCK interface</b>	<b>65</b>
14.1	Background . . . . .	65
14.2	Using iROCK . . . . .	65

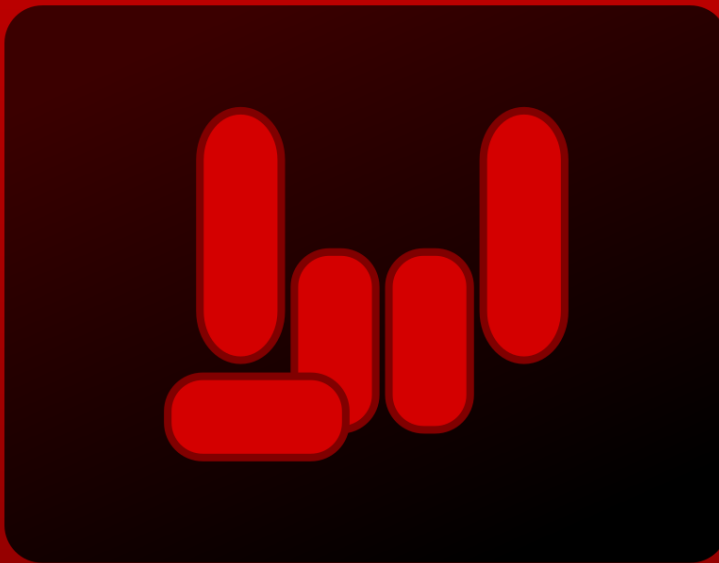
<b>III</b>	<b>Applying the ROCK</b>	<b>67</b>
<b>15</b>	<b>A ROCK workflow</b>	<b>69</b>
15.1	A basic ROCK workflow . . . . .	69
15.2	An extensive ROCK workflow . . . . .	72
<b>16</b>	<b>A Beginner's Guide to the Reproducible Open Coding Kit</b>	<b>77</b>
16.1	Welcome! Step-by-step, no prior knowledge of R needed! . . . . .	77
16.2	Setting up the Basics . . . . .	78
16.3	Data Preparation . . . . .	95
16.4	Coding and Segmentation . . . . .	103
16.5	Aggregation and Analyses . . . . .	106
<b>17</b>	<b>Using the ROCK for Epistemic Network Analysis</b>	<b>107</b>
17.1	Starting point . . . . .	107
17.2	Planning coding . . . . .	108
17.3	Planning Segmentation . . . . .	109
17.4	Designating Sources and Cases . . . . .	111
17.5	Designating Attributes . . . . .	111
17.6	Coding and Segmentation . . . . .	113
17.7	Merging Coded Sources (if necessary) . . . . .	114
17.8	Creating Networks . . . . .	115
<b>18</b>	<b>Using the ROCK for Cognitive Interviews</b>	<b>117</b>
18.1	Analysing cognitive interviews using the Shiny ROCK Beryl . . . . .	118
18.2	Analysing cognitive interviews using the Shiny ROCK Amethyst . . . . .	118
<b>19</b>	<b>Using the ROCK for a Qualitative Network Approach</b>	<b>121</b>
19.1	Customizing network appearance . . . . .	121
<b>20</b>	<b>Using the ROCK for Decentralized Construct Taxonomies</b>	<b>123</b>
20.1	The importance of clear coding instructions . . . . .	123
20.2	Introduction to Decentralized Construct Taxonomies . . . . .	124
20.3	Creating a DCT . . . . .	124
20.4	Coding with DCTs . . . . .	127
20.5	Analysing DCT-coded sources . . . . .	127
<b>IV</b>	<b>Appendices and references</b>	<b>129</b>
<b>21</b>	<b>References</b>	<b>131</b>



<https://rockbook.org>

# THE ROCK BOOK

The Reproducible Open Coding Kit



Szilvia Zörgő & Gjalt-Jorn Peters



This is the Reproducible Open Coding Kit book, a living<sup>1</sup> Open Access<sup>2</sup> book. The Reproducible

<sup>1</sup>A living document is a document that can be updated over time. Conventional books have editions; living books can be updated in smaller steps.

<sup>2</sup>Open Access means that it is free. Specifically, the license attached to this book for now is the CC-BY-NC-SA

Open Coding Kit, the ROCK, is a standard for qualitative research. It was developed to facilitate reproducible and open coding, enabling explicit documentation of the various analysis steps without compromising the flexibility that lends much of the qualitative research its strength.

At its core, the ROCK specifies conventions for including codes and attributes in plain text files. This standard enables coding and reading qualitative data without the requirement of any software other than a plain text file editor. Simultaneously, it allows the development of software to, for example, further process those files to support specific analyses, or to provide a graphical user interface to facilitate coding. This book will discuss two such implementations of the ROCK. The first is the **rock** R package, an R package originally developed to facilitate using the ROCK standard to support Epistemic Network Analyses, but since then extended to also use the ROCK for cognitive interviews and to work with decentralized construct taxonomies (DCTs) as produced by psyverse. The second is iROCK, a rudimentary graphical user interface to code text files using the ROCK.

The ROCK helps qualitative researchers navigate the imperatives created by Open Science and the General Data Protection Regulation. On the one hand, because no proprietary software is required, researchers retain complete control over the data they process. This means that they do not need to enter into data processing agreements with third parties if they process identifiable data. On the other hand, if they work with anonymized data, because the data are stored in plain text files, they can easily be shared with other researchers, who can then easily re-run (or adapt and re-run) the analyses.

Note that this book is mostly written to cover reproducible coding of qualitative data using the ROCK, although it does cover some basic underpinnings of qualitative methods in Part I. Chapter 1 will start with a brief introduction of a number of qualitative approaches which are then discussed more in detail in the other Chapters in Part I. Part II will continue, based on this introduction, with Chapter 10, which will establish a vocabulary. At the hand of this vocabulary, Chapter 11 will introduce the ROCK standard. Chapter 12 will discuss the **rock** R package, and Chapter 14 will discuss the iROCK interface. Chapter 15 combines all this in an example of a ROCK-based workflow. Chapter 17 concerns applying the ROCK to Epistemic Network Analysis, Chapter 20 concerns applying the ROCK to work with decentralized construct taxonomies, and Chapter 18 concerns applying the ROCK to work with cognitive interviews. Because this is a living book, more chapters may be added.

Impatient readers may want to skip ahead to Chapters 15, 17, 20, and 18, if those match one of their use cases. At present, this book is still under development: the first complete edition has not yet been released. Since the **rock** R package used in this book is also under active development, at this point you may want to install the development version of the **rock** R package (see Chapter 12).

To cite this book, you can use:

Zörgő, S. & Peters, G.-J. Y. (2019) The ROCK book (1st Ed.). <https://doi.org/10.5281/zenodo.3571020>



## Part I

# Qualitative Research



# Chapter 1

## Qualitative research

Disciplines that directly or indirectly study the human psychology, such as anthropology, sociology, criminology, and psychology, face the problem that the studied objects are not directly observable (and often are not natural kinds<sup>1</sup>). The methods with which the indirect observations that are used instead are collected can be divided into two types: quantitative and qualitative methods.

Quantitative methods are comparable to most measurement instruments, and aim to map unobservable variables unto a quantitative scale. The means through which this occurs is called operationalization of those variables, and without a valid operationalization, quantitative methods cannot be applied. Qualitative methods do not require such quantification. This is simultaneously a strength and a weakness.

[...]

---

<sup>1</sup>Explain plus link to Eiko's commentary.



## Chapter 2

# Quantitative Ethnography

Quantitative ethnography (QE) is a nascent field aiming to establish a unified, quantitative – qualitative research methodology. This endeavor entails the formulation of novel concepts, negotiation of terminology, and developing new techniques that serve a wide variety of research initiatives in multiple disciplines. QE research involves or is conducted on Discourse data; this may include data in the form of spoken or written speech (interview transcripts, log files, social media data, etc.), annotations on observations (field notes, structured observation, etc.), and visual data (video recordings, photovoice, etc.). Other data may be collected as well, serving as “metadata” or characteristics of the data providers, data collection, or the data itself (e.g., timestamps, locations, sociodemographic data). Metadata can be used to group data providers and their discourse, enabling a wide variety of analyses. In order to represent various types of data in a single, unified dataset in a human and machine-readable format, qualitative data needs to be quantified. This is typically achieved through discourse segmentation and coding. This unified dataset can then be employed for performing various analyses and modelling complex interactions. Zörgő et al. (2022)

Several tools and techniques have been developed under the auspices of QE, aiming to facilitate a part of the process described above or to enable various analyses using the generated dataset. For example, nCoder is a platform that can be utilized to develop, refine, validate, and implement automated coding schemes. Epistemic Network Analysis (ENA) has been the flagship tool within QE, enabling researchers to explore and model salient patterns in their data with network graphs. ENA depicts the structure of connections among codes in discourse by calculating the co-occurrence of each unique pair of codes within a designated segment of data, and aggregates this information in a cumulative adjacency matrix. ENA represents this matrix as a vector in high-dimensional space, which is then normalized to quantify relative frequencies of co-occurrence independent of discourse length or simple volume of talk. In this process, network connection strengths are transformed to fall between zero and one, and ENA then projects the networks as points into a low-dimensional space using rotations, such as singular value decomposition (SVD), which maximizes the variance explained. This analytical tool hence provides two, coordinated representations of the unified dataset: 1) network graphs, where the nodes in the model correspond to the codes in the discourse and the edges represent the relative strength of connection among codes, and 2) ENA scores, or the position of the plotted points on each dimension of the plotted points for units of each network in the low-dimensional space. Network models are thus constructed based on researcher decisions made when creating the qualitative data table (e.g., operationalization of segmentation and coding) and decisions regarding network parameterization. Zörgő et al. (2022)

Such unified methods and data modelling tools enable the researcher to ask both quantitative and qualitative questions about their data, be able to represent both aspects of data in a single dataset, and develop quantitative models that inform qualitative understanding and vice-versa.



## Chapter 3

# Qualitative methods in psychology

Within psychology, there are broadly two ways in which qualitative methods are used, one fundamentally inductive, the other using a combination of inductive and deductive approaches. First, if (almost) no theory exists about a subject, qualitative data can be collected to contribute to the creation of a large evidence base in which patterns can be identified that can then give rise to theory. Second, if theory does exist, the constructs posited by those theories often have content that differs for different populations, contexts, and behaviors. Once a theoretical construct has been clearly defined, it is possible to derive guidelines for eliciting the construct's content in a given population, context, or otherwise relevant circumstance.





## Chapter 4

# Cognitive interviews

Cognitive interviewing is a versatile method that can be helpful to explore how people perceive, process, and respond to stimuli. It is a popular approach to study UI/UX (user interface design and user interaction and experience) in industry, as well as for exploring how people perceive questions and questionnaires in marketing. In the social sciences, it is often used to study cognitive validity or “actual” validity.

Therefore, before going into cognitive interviews themselves, we will briefly explain cognitive validity in the context of measurement instruments and manipulations. We will then proceed with explaining cognitive validity, after which we will deepen the discussion by delving into response processes, narrative response models, and finally, explaining how cognitive interviews can be used to study the validity of a measurement instrument or manipulation.

### 4.1 Cognitive validity, measures, and manipulations

Cognitive validity usually refers to whether people interpret instructions, questions and questionnaires as intended. Specifically, cognitive validity refers to whether participants interpret the procedures, stimuli, and response registrations that form a measurement instrument (or manipulation) as intended.

Examples of manipulations are elements of behavior change interventions (e.g., the application of a behavior change principle such as modeling or persuasive communication), ingredients of psychotherapy (e.g. an approach to help people with reattribution or protocol elements designed to foster trust in the therapy-client relationship), or manipulations as used in psychological experimental studies (e.g. stimuli such as sound fragments that should induce a feeling of stress, a procedure designed to temporarily increase participants’ self-esteem, or a set-up designed to produce the smell of apple-pie).

Examples of measurement instruments are questionnaires (e.g., an experiential attitude scale designed to measure people’s feelings towards hand washing, a survey aiming to map out people’s perceptions of traffic safety, or personality indices) or response latency tasks (e.g. the implicit association test).

Both manipulations and measurement instruments consist of procedures and usually stimuli, and measurement instruments also specify how to register participants’ responses. For the manipulations and measurement instruments to be valid for a given population in a given context, usually a first requirement is that these constituent procedures, stimuli, and response registrations are interpreted as intended. Cognitive interviews are a method to study whether this is the case.

We will use the term “item” to refer to one single element of a measurement instrument or manipulation. A single measurement instrument item corresponds to the registration of a response. For example, in a questionnaire, each question is an item, and in a response latency task, each trial is an item. In a manipulation, each combination of stimuli and procedural information that is comprehensible on

its own (or more accurately, that is capable of producing the desired effect on its own) is an item. For example, manipulation items can be single auditive stimuli (audio clips or songs), videos, images, paragraphs of text, or (elements of) protocols. Often, manipulations will be one item, intended to be interpreted in unison.

In the rest of the chapter, we will use questionnaire items as examples. Remember that everything also applies to other types of measurement instruments (such as response latency tasks) as well as to manipulations. Anything that was designed to produce some specific effect or response can be explored to some degree with cognitive interviews.

## 4.2 Response processes

A useful concept when thinking about how people perceive items (i.e., procedures, stimuli, and methods to register responses) is response processes. For measurement instruments, people's response processes are the processes that produce the response that is then registered ("measured"). For manipulations, response processes are the processes produced in response to the item (not necessarily resulting in a response that is registered).

Response processes usually start the same way: with people's perception of an item. After the item is perceived, people interpret what they perceived and further process that information. Cognitive validity refers to how "correct" these response processes are.

As an example, let us take the infamous Pickle Fanaticism Scale. Its first iteration has 33 items in three subscales ("Strong desire to eat pickled products", "Extreme liking of pickled products", and "Feels the need to evangelize about the benefits of pickles"). Below are three items, one from each subscale:

1. I often contemplate the role of pickles in a post-modern society
2. I always eat pickles for breakfast
3. I recommend pickles to people I know

The response scale is the same for each of the 33 items: "Strongly Disagree" (1); "Disagree" (2); "Neither agree nor disagree" (3); "Agree" (4); and "Strongly Agree" (5).

Each item can be considered to be formed by six stimuli: the question texts (the three sentences) and each of the five response options. Each of these stimuli, as well as the procedure determining how they are laid out on the page, can be interpreted differently. These interpretations drive further processing that ultimately results in a response that is registered.

With respect to cognitive validity, this means that for an item to have cognitive validity, each of these stimuli must be interpreted as intended. For example, participants may have different definitions of "often" (as in the first item; e.g. "more than half the time" or "more than two-thirds of the time"), "post-modern society" (also in the first item; a term that quite a lot of people may not be familiar with), "always" (second item; e.g. "100% of the time" or "90% of the time or more"), and how broadly they define the group of "people I know" (i.e. whether this includes distant acquaintances or not). If the stimuli are not interpreted as intended, participants effectively answer a different question.

In an influential book about the psychology of survey responses, Tourangeau, Tips and Rasinski (2000) describe a model with four components: comprehension, retrieval, judgment, and response. Each of these is subdivided into specific processes: comprehension contains attending to questions and instructions; representing a logical form of the question; identifying the question focus (i.e. what is being asked); and linking key terms in the question to relevant concepts. The retrieval component is subdivided into generation of a retrieval strategy and cues; retrieving memories; and filling in missing details. Judgment consists of assessing the completeness and relevance of the memories; drawing inferences based on accessibility; integrating the material; and making an estimate based on partial

retrieval. Finally, the response component consists of mapping judgment onto the response options and editing the response.

Because many different types of measurement instruments exist, there is no single response process model that is useful across the board as a model for how people process questions. As a consequence, multiple response process models exist. However, this generic model is a useful scaffold when thinking about what happens when people respond to a survey. In addition, it can be used to organize the results of cognitive interviews, discussing which items exhibit problems with each of the four components in this model (comprehension, retrieval, judgment, and response).

## 4.3 Cognitive interviews

Cognitive interviews use two methods to explore participants' response processes: the Think Aloud method and Probes.

### 4.3.1 The Think-Aloud method

The Think-Aloud method does pretty much what it says on the tin: participants are asked to think aloud. This is of course less trivial in practice: thinking aloud is (for most people) not natural, so participants need some practice and often have to be reminded a few times.

When using the think-aloud method, it is important that the interviewer limits themselves to reminding the participants to think aloud, for example by asking “what are you thinking?”

Because during the actual cognitive interview, the interviewer cannot say much else, training participants in thinking aloud is important. To train them, prepare a number of questions or scenarios you can use. For example, Willis (1999) suggests the following:

“Try to visualize the place where you live, and think about how many windows there are in that place. As you count up the windows, tell me what you are seeing and thinking about.”

*Willis, 2009, p. 4*

For some participants, multiple such tasks may be required before they get the hang of it, so so you want to prepare several different tasks to have them ready when required.

If the cognitive interview is not recorded, during the cognitive interview, the interviewer makes notes of their interpretations of the participants' response processes on the basis of their thinking aloud. Of course, if the interview *is* recorded, notes can still be helpful to provide details that will not be captured on the recording.

### 4.3.2 Verbal probing

Benefits of think-aloud approaches are that there is minimal risk of intervening with participants' ‘natural’ response processes. On the other hand, despite the training in thinking aloud, participants can vary quite a bit in how comprehensively they think aloud. In addition, they will not generally elaborate on specific issues that may be relevant to you as a researcher.

Therefore, the think-aloud approach is often combined with verbal probing. Probes are questions that you prepare to, well, probe specific issues. For example, you may anticipate that participants may heterogeneously define the group of “people they know”, as in the question “I recommend pickles to people I know”. If that is the case, you may want to prepare a series of probes to explore these interpretations. For example, you could prepare probes such as “When you answered this question, what did “people I know” mean to you?” or “To you, what determines when you know somebody?”.

There are more extensive guidelines in ‘Cognitive Interviewing: A “How To” Guide’ by Willis (1999), available at <https://bit.ly/wbm-willis-ci>. A comparison of different types of verbal probes is available in ‘Which probes are most useful when undertaking cognitive interviews?’ by Priede, Jokinen, Ruuskanen & Farrall (2014), available at <https://bit.ly/wbm-priede-ci-probes>.

## 4.4 Common cognitive interview coding schemes

A number of commonly used cognitive interview coding schemes exist [some are listed below; for more details, see Peterson et al. (2017) and Woolley et al. (2006)]. Whether you use existing coding schemes, develop your own, use existing schemes with more detailed codes added relating to your specific response models, or even have multiple coders code using different coding schemes is ultimately a subjective, scientific, and pragmatic consideration. Like for all decisions you take in any scientific endeavour, the most important thing is to clearly, comprehensively, and transparently document your decision and the underlying justification.

### Peterson, Peterson & Powell

- understanding** Is the item wording, terminology, and structure clear and easy to understand?
- retrieval** Has the respondent ever formed an attitude about the topic? Does the respondent have the necessary knowledge to answer the question? Are the mental calculations or long-term memory retrieval requirements too great?
- judgment** Is the question too sensitive to yield an honest response? Is the question relevant to the respondent? Is the answer likely to be a constant?
- response** Is the desired response available and/or accurately reflected in the response options? Are the response options clear?
- content\_adequacy** Do all of the items combined adequately represent the construct? Are there items that do not belong?

### Levine, Fowler & Brown

- comprehension** Items with unclear or ambiguous terms, failed to understand the questions consistently.
- knowledge** Items for which respondents lacked information to answer a question.
- inapplicable** Items measuring construct that are inapplicable for many respondents (e.g. made assumptions).
- construct** Items failed to measure the intended construct.
- subtle** Items making discriminations that are too subtle for many respondents.
- general** Several other general issues associated with the development of a questionnaire.

### Willis, 1999

- clarity** Problems with intent or meaning of a question.
- knowledge** Likely not to know or have trouble remembering information.
- assumptions** Problems with assumptions or underlying logic.
- response\_categories** Problems with the response categories.
- sensitivity** Sensitive nature or wording/bias.
- instructions** Problems with introductions, instructions, or explanations.
- formatting** Problems with lay-out or formatting.

## 4.5 Coding cognitive interviews

For more information on coding cognitive interviews, please see chapter Using the ROCK for Cognitive Interviews, chapter 18 in this version of the book.

## 4.6 Towards test validity: response models

Cognitive validity is a useful concept even when thinking about items that are used to, for example, collect information about people’s age, gender, or education level. However, in psychology, items are often used to measure psychological constructs. In such cases, cognitive validity is still important, but mostly as a first prerequisite for “regular” validity.

This “regular” validity is a complex concept, and various definitions and approaches have been proposed over time (see e.g., Borsboom et al., 2004). It is generally defined as whether the item (or set of items aggregated into a measurement instrument) measures what it is supposed to measure. Although this is the commonly used definition, the commonly used procedures for judging validity do not align with that definition. These issues are discussed in depth in (Borsboom et al., 2004), (Borsboom et al., 2009), and this disconnect is clearly illustrated by (Maul, 2017).

One solution is to abandon the construct-instrument correspondence usually seen as the core of validity, and define validity as a function of the interpretation and use of an item or measurement instrument (Kane, 2013). However, (Borsboom et al., 2009) argue that this also seems untenable (Borsboom et al., 2009), instead proposing what they call “test validity” as a solution. This test validity is in essence the validity people talk about when talking about validity without specifying a type: whether a measurement instrument measures what it is supposed to measure.

Borsboom, Cramer, Kievit, Zand Scholten, and Franic (2009) argue that almost all measurement models are reflective measurement models: put simply, they assume that item scores reflect an underlying latent construct. They further argue that the model of how the construct of interest causes these scores is central to validity, or as they put it, “In our view, in fact, establishing the truth of such a model would *clinch* the question of validity.” (Borsboom et al., 2009, p. 155). They argue that research into validity should concern itself with response processes to establish, in short, how a given item or measurement instrument works: “If successful, such a research program therefore solves the problem of test validity, because it by necessity becomes clear what the items measure and how they measure it. And that is all there is to know regarding validity.” (Borsboom et al., 2009, p. 155)

As an instrument to think about and explicate one’s theory of how an item or measurement instrument work, the concept of response models can be helpful. We define a response model here as the *intended* process leading to the registration of the participants’ responses and starting with their exposure to the stimuli and procedures that, together with the response registration procedure, constitute an item. An item is a distinct element of a measurement instrument or manipulation: these typically (but not necessarily) consist of multiple items. Although the logic underlying response models as means to specify what happens in between when somebody perceives stimuli and produces a response applies to manipulations as well as to measurement instruments, in the remainder of this chapter we’ll discuss measurement instruments as they explicitly contain a procedure for registering responses.

The response models describe how the construct that an item is designed to assess causes the variation in the item’s scores (if the item performs as it’s supposed to, i.e. if the item is valid; see Borsboom et al. (2004) and Borsboom et al. (2009) for more background and how this approach contracts with Kane (2013)’s argument-based approach). It describes how more fundamental psychological processes are invoked, for example how the relevant reflective and/or reflexive, cognitive and/or affective, deliberate and/or automatic constructs, such as mechanisms, processes, or representations, ultimately produce the response that the item registers.

Note that the type of response model can differ as a function of one’s ontological and epistemological perspective. From more constructivist perspectives, the response model may involve shared construction of meaning; perspectives tending towards realism might lean more heavily on attention or memory processes; and if one entertains an operationalist perspective, response models might be exceedingly rudimentary (though admittedly, researchers with that perspective would probably not engage in cognitive interviews in the first place).

## 4.7 Response processes revisited

As discussed above, participants' response processes are the description of what happens as they perceive, interpret, and process an item and ultimately produce the response that is registered by the item's response registration procedure. From a validity perspective, then, these response processes ideally closely reflect the item's response model.

The form of a response process is typically quite different from the form of the response model. The former is often derived from participants' verbal descriptions that express the results of introspective efforts; whereas the latter is often a description of the involved theoretical constructs and mechanisms (see the previous section). For example, the latter can contain automatic or unconscious processes, which would, almost literally by definition, be unavailable to introspection. Therefore, complete overlap between response processes and the desired response models may be impossible.

## 4.8 Selecting response model parts

Therefore, not all parts of the response models for all items can always be verified using cognitive interviews; other methods may have to be invoked, such as experiments where items are manipulated to verify parts of the item's response model. This means you will have to first decide how each part of the response model can be verified, and then for the cognitive interviews, select those parts of each item's response model that in fact lend themselves to verification with cognitive interviews.

Once you made this selection, you have a list of response model parts for each item. Often, the response models for a set of items that belong to the same measurement instrument will overlap. To illustrate this, we will give two examples of common situations. First, sometimes you assume that two items are so-called "parallel items". These are items that you assume measure the exact same thing in the exact same way and assuming a reflecting measurement model. Such items are, for all practical purposes, interchangeable — and in such cases, the response models will be identical.

Second, sometimes you have two items that measure very different things. For example, one item is designed to measure somebody's income, and a second item is designed to measure somebody's education. These can both be part of a measurement instrument for social-economic status that assumes a formative measurement model. The response models for these two items will be quite different.

Once you made this selection, you should have the following:

- for each item, a response model;
- for each part of each response model, a decision as to whether you think it's feasible to study whether those parts of people's response processes are consistent with the corresponding parts of the item's response model;
- for each part of each response model, in which other response models it occurs.

You need one more piece of information before you can move towards specification of the coding schemes and the preparation of prompts: the response process spectrum.

## 4.9 The response process spectrum

For every part of each response model you want to verify, specify the corresponding response process spectrum. This is, as far as you can think of, a list of all possible alternatives to the response model. For example, if your response model contains a step "the person visualises the front wall of their house", for example in an item measuring the number of windows in somebody's house, a potential response process spectrum could be:

- the person visualises the front wall of their house (i.e. the response model)
- the person visualises the back wall of their house
- the person visualises a side wall of their house
- the person does not visualise anything

During the actual cognitive interview, you will likely discover that people’s response processes deviate from the response model in ways you couldn’t imagine beforehand, and that’s ok. The main purpose of this step is to help you get an idea of the kinds of things you’ll want to spot in each part of each response model.

Once you produced the response process spectrum for all response model parts for all (unique) response models, you can start compiling your coding scheme.

## 4.10 Coding schemes based on response spectra

Based on the response spectra for each part of each response model, you can now produce codes that you will use to code the notes (or maybe transcripts) of your cognitive interviews. These codes will be the “glasses” through which you will see your results: although you can always add new codes during the coding phase, in general, it is easy to miss things for which you did not prepare a code.

For each part of each response model, think of a brief code that tells you how people’s response processes look. The coding scheme can be hierarchical: you can have “sub-codes” or “child codes” to organize your codes. For every code, designate a code identifier: a unique string of characters consisting only of lower case letters (a-z), upper case letters (A-Z), digits (0-9), and underscores (\_), and always starting with a letter. If you have hierarchical codes, you can indicate the hierarchy using a greater-than sign (>).

Examples of valid codes are:

- `memory>recall`
- `attention`
- `forgot>early_childhood`
- `forgot>late_childhood`

Finally, once you have your coding scheme, you can craft your prompts.

## 4.11 Preparing prompts

Prompts are specific things you can ask participants during the cognitive interview designed to elicit expressions that are informative as to specific parts of their response processes. Designing these is relatively straightforward once you have your coding scheme: you have to think about which questions you can ask that are likely to lead to answers that you can then code with specific codes pertaining to specific parts of the response process.

Of course, if you manage to formulate prompts that can cover multiple parts of participants’ response processes, that’s more efficient. Therefore, asking open-ended questions (e.g., “why did you provide that answer?”) is a popular approach. However, sometimes closed questions can be very efficient to quickly check whether people did a given thing (e.g., “to arrive at this estimate, did you visualise the front wall of your house?”).

The product of this step will be a list of prompts, that you sort in the order in which the items will be presented to participants. You may want to designate unique identifiers to the prompts (e.g. numbers, letters, or a combination of these) to structure the notes you take during the cognitive interview, or even enter your notes into a file that already contains the prompts.

#### 4.11.1 A note on using existing coding schemes

Using existing coding schemes (see the section above) has advantages and drawbacks. A salient advantage is that if you use an existing coding scheme, you don't have to map out the response models and response process spectrums for each item. This saves you a lot of time and potentially frustration and uncertainty if you don't know much about the relevant response models. Another advantage is that using an existing coding scheme facilitates comparison of item performance (and so, measurement instrument performance) over different cognitive interviews studies.

A big disadvantage is that if you use an existing coding scheme, you don't have to map out the response models and response process spectrums for each item. Those exercises force you to think long and hard about the assumptions underlying each item's validity (and so, the validity of your measurement instrument), and skipping this make it more likely you miss problems. A second disadvantage is that it is harder to see how to improve the items, since the results of your cognitive interview will be very generic; they will not point to specific parts of the response models.



# Chapter 5

## Initial codes

In 1974, Richard Feynman gave a commencement address at Caltech, a university in the United States, where he not only coined the term “cargo cult science”, but also formulated a very succinct definition of scientific integrity:

The first principle is that you must not fool yourself – and you are the easiest person to fool. So you have to be very careful about that. After you’ve not fooled yourself, it’s easy not to fool other scientists. You just have to be honest in a conventional way after that.

In research, fooling yourself is excessively easy due to what has been referred to as “researcher degrees of freedom” (Simmons et al., 2011) or “the garden of forking paths” (Gelman & Loken, 2014): conducting a study involves making hundreds of decisions, varying from trivial to critical, and each of these decisions is an opportunity to fool yourself unintentionally (or, for that matter, intentionally).

Since humans have a lot to gain from effectively fooling ourselves much of the time, it’s very hard to realise when you’re fooling yourself. Doing research with scientific integrity, therefore, requires availing oneself of an array of supporting tools. In qualitative research, one of these tools is the practice of carefully developing a set of initial codes.

At first glance it may seem like developing an initial code set makes sense when planning to engage in deductive coding, but less so when planning to use a more inductive (or ‘open’) approach. However, to avoid fooling yourself, it is always wise to develop an initial set of codes. This practice can be seen as an operationalization of your positionality: exercising reflexivity to explicate your prior ideas about your study in an initial set of codes helps you to avoid fooling yourself. After all, those prior ideas will influence your coding regardless of whether you’re transparent about them or not, and if you didn’t explicitly elaborate and document them a priori, it is easy to fool yourself into thinking your ultimate coding structure represents insights obtained from the data, when in fact it represents your presupposed ideas instead.

Facilitating transparency about your pre-existing expectations and assumptions isn’t the only benefit gained by establishing your initial set of codes. A second benefit is that the process of coding becomes more consistent: every coding decision becomes an explicit interaction between the existing code structure and the data fragment(s) you are evaluating (see chapter Coding, chapter 7 in this version of the book). The third benefit is closely related: this more uniform approach to coding makes it easier to comprehensively document and justify your decisions. This, in turn, facilitates reflexivity again.

### 5.1 Code organization

One of the decisions you will have to take when you want to code qualitative data is how your codes will be organized. Codes can be organized in any way you want, provided you have a system available that

supports that organization. A consequence of this (i.e. the availability of such systems constraining the code organizations that are used) is that in practice, only three organizational modes are common. The mode you select is determined by the organizational structure that you assume exists in the patterns you are looking for in the data. This organizational mode, therefore, is simultaneously a manifestation of your prior beliefs about your research topic, and something of a mold that will shape your results.

### 5.1.1 Flat code organization

Sometimes, codes are simply not organized at all. You then attach codes to fragments of data and leave it at that. You can still refine the codes by working iteratively, but this results in either replacing existing codes attached to a data fragment or attaching additional codes. You select this mode of code organization if you assume that the patterns you are interested in do not exhibit a structure mimicked by an available organizational mode.

Because a flat code organization imposes no structure on your data, you do not risk believing you discovered a certain structure in the data while in fact it is just the structure your codes were cast into by the organizational mode you chose. However, the drawback of a flat code organization is that if there *is* an intrinsic structure to the data that, when reflected in your codes, would help you understand the data better, you will be less likely to discover it.

### 5.1.2 Hierarchical code organization

The most common mode of code organization is hierarchical organization. This means that codes are clustered within other codes. This organizational mode fits well with many qualitative paradigms and their approaches where the code structure is refined through iterative coding. Each iterative round then often consists of adding more specific codes as ‘children’ of the codes that the round started with.

The concept of codes representing more specific manifestations of overarching codes is intuitive to many people. In addition, this organizational mode also lends itself well to a combination of deductive and inductive coding: the initial code set then resembles a shallow hierarchical code tree, and these codes become the parent codes of the inductive codes that are added.

### 5.1.3 Network code organization

Organizing codes in a network is both more complicated and more versatile than the other two organizational modes. The tree structures used for hierarchical code organizations can be represented using networks, and so can flat code structures. However, networks can also be used to represent causal relationships expressed in a dataset, or specific types of structural relationships. Therefore, choosing this organizational mode means you have quite clear ideas about the structure of what you are looking for, and that entails both the highest potential for introducing bias and the highest potential for accurate representation of your data.

A clear benefit of the network organizational mode is that the relationships between two codes can vary. In a hierarchical code structure, the relationship between a parent code and a child code is itself normally not coded: the meaning of that parent-child relationship is therefore implicit, and usually the same for the entire code tree. If it is not the same for the entire tree, this risks introducing some ambiguity: for example, in some branches parent-child relationships may represent a structural relationship (e.g. “children are parts of their parents”), whereas in other branches parent-child relationships may represent a specifying relationship (e.g. “children are more specific concepts than their parents”).

When using network coding, every relationship is explicitly coded. This clearly captures the coder’s belief that a data fragment expresses, for example, a causal relationship (“coding causes headache”), a structural relationship (“qualitative research is a type of research”), or an attributive relationship (“qualitative data is very rich”).

Perhaps the biggest benefit is that the structure of the codes is not constrained in any way. A code (e.g. “coding”) can simultaneously be a cause of another code (“causes headache”), have an attribute (“takes time”), and have parts (“choosing a code identifier”), and this can all be represented in the code structure with high fidelity. This means that generally, using the network organizational mode means that coding stays closer to the data: it requires less imposition of a given (e.g. hierarchical) structure.

A drawback related to this versatility is that networked code structures can quickly become complex. In addition, once multiple sources have been coded, the resulting code structures can be hard to combine. For example, in one source, the causal relationship “coding has a positive causal relationship with headache” can be coded, whereas in another source, the causal relationship “coding has a negative causal relationship with headache” can be coded. Some people may get a headache from coding, but for others, coding may help to ameliorate headaches. Reconciling such code structures can be a hard task.

#### 5.1.4 Combining and changing organizational modes

If your situation calls for it, you can also combine multiple organizational modes. In addition, if you initially suspected that a given organizational mode would fit well, but during coding, you discover that you were wrong, you can always switch. That does entail some costs, though, which become higher as you switch later in the process: you will have to do more and more recoding, after all. Therefore, it pays to reflect thoroughly on the organizational mode with which you start: this is not a decision you should take lightly.

## 5.2 Compiling your initial set of codes

Once you decided on the initial organizational mode for your initial code set, you can start thinking about the codes populating it. In this process, it helps to determine where you are located on the spectrum from the intention to work fully deductively to the intention to work fully inductively.

On the one extreme of this spectrum, where you aim to work fully deductively, you only apply codes that you developed in advance. The code structure you end up with is identical to the code structure you started with. In this scenario, you already have one or more theories or other guiding principles that gave rise to this initial code structure, and you’re not interested in adapting that structure based on what you encounter in the data - instead, you’re usually mainly interested in the fragments of data you end up labeling with every code. At this end of the spectrum, developing an initial code structure is relatively easy, since explicitly specified theories or guiding principles already exist.

On the other extreme of this spectrum, where you aim to work fully inductively, your situation is opposite: you try to be as blank a slate as possible, and represent whatever patterns you find in the data as well as possible. However, nobody who has developed the competence to conduct a qualitative study, or to code data, is truly a blank slate: people have many representations of the world, and these influence how they interpret fragments of that world such as qualitative data. At this end of the spectrum, it can be quite hard to develop an initial code structure, as this will require considerable self-interrogation so as to unearth the various relevant expectations and preconceptions that form the breeding ground for the produced codes.

As the name implies, these two extremes are quite rare. Usually researchers find themselves in one of the infinite positions on the spectrum in between. To offer guidance to develop an initial code structure wherever you may find yourself, the approach from each extreme will be discussed, so you can take the elements that suit your specific situation.

### 5.2.1 Developing initial codes based on extant specifications

If you are working from extant specifications, such as a theory (or several) or another collection of ideas that will guide your coding, the following steps can be useful to develop your initial set of codes. The order in which these steps are presented isn't meant to be prescriptive; the process of fleshing out your initial code structure is often iterative and somewhat messy.

- Think about the concepts and relationships defined in the theory and decide for which ones you want to identify corresponding data fragments.
- Once you have that list ready, discuss it with everybody in your project team (i.e., all collaborators) as well as relevant external stakeholders (e.g., representatives of relevant groups to your study, such as the population you're interested in). Make sure everybody agrees on this initial list of concepts.
- For everything you want to code, write up a comprehensive preliminary definition. "Comprehensive" here means that it should be so specific and accurate that discussing it with others will reveal (potentially subtle) differences in each individual's definition.
- Once you have comprehensive preliminary definitions, discuss these with all collaborators and external stakeholders. This process will often result in decisions to split or merge or otherwise reconfigure the concepts and relationships, and can take quite some time.
- At this point, you have a list of things you want to code, and your team has shared definitions of what these things are. The next step is to develop coding instructions – this is discussed in the section below.

### 5.2.2 Developing initial codes based on personal preconceptions

If you are working without pre-existing specifications to guide your coding, this means the codes you will apply to the data will be the product of both the data and your mind. You see the world in a specific way based on your life so far, and this produces the lens through which you will perceive the data, and so defines the language and conceptual elements you will use to devise codes.

Therefore, when developing initial codes, your aim is to use the fact that a study has a relatively narrow contextual scope to reflect on your perspective and how this will guide your inclinations as you code the data in your study.

- Think about the kinds of things you'll expect to find in the data. Try to imagine prototypical expressions of those things, and then think about the categories these would fit into.
- Give every category you come up with a label and a description. In this process, you may reconfigure your categories, as describing them more in detail may reveal similarities and differences that were not immediately obvious. The categories you end up with represent your preliminary initial set of codes.
- If you will be the only person coding, you can now develop coding instructions. If multiple people will code, you have to decide how to proceed. If you want to retain the differences between coders as would be the case if each would do the project individually, each coder develops their own coding instructions. However, if you want the coders to start from the same initial perspective, first discuss the preliminary sets of codes each prospective coder can come up with. Through these discussions, merge and reorganize the code sets into one preliminary code set that you all feel represents the conceptual starting point for the project well. Then, you can move on to develop coding instructions to accompany these codes.

## 5.3 Developing coding instructions

Having an initial list of codes isn't enough to be able to start coding. The process of coding involves many, many decisions (see chapter Coding, chapter 7 in this version of the book), and so offers many opportunities to fool yourself. Therefore, you will need to support your future self (or other coders) to avoid that. A valuable tool to this end is a clear coding instruction.

Coding instructions bridge the gap between what a code represents on a conceptual level and how you recognize expressions of it in qualitative data. It is not uncommon that, as you develop your coding instructions, discussing them with others will lead to revision of the concepts you want to code. Because coding instructions are more concrete than definitions, often this reveals differences in mental models that are easy to neglect to flesh out when discussing conceptual definitions. Developing coding instructions is therefore a useful tool to make sure your initial code set is sufficiently well described.

### 5.3.1 Edge cases

An important part of coding instructions is the explicit description of edge cases. Coding instructions often center around the core of the relevant code. This is helpful to identify data fragments that clearly represent expressions of the relevant code, but less helpful when encountering data fragments where matters are less clear-cut.

This is where edge case descriptions come in. Edge cases are data fragments that have certain characteristics matching the relevant code, but other characteristics that could justify not applying that code, or applying a different code instead. Thinking through such edge cases in advance and explicitly describing them helps you to delineate your initial codes. It helps you to clarify, to others as well as to yourself, what exactly you mean with each code.

Edge case descriptions also allow you to specify under which conditions one code should be applied, and under which conditions another code should be applied instead. Thinking about where one code ends and another starts and documenting this in cross-references between your codes is very helpful for others and future you to conceptualize your codes in relation to each other.

### 5.3.2 Examples

As discussed above, while code labels and descriptions are conceptual, coding instructions are more concrete and bridge the gap to expressions you can encounter in your data. Examples are the most concrete parts of coding instructions, and can be very helpful both for yourself and your collaborators to test whether you're on the same page regarding what codes represent, and for others to wrap their heads around what exactly should and should not be coded with a code.

In practice, while you think about and discuss your initial code book, it is likely you will already do this at the hand of real or fictional data fragments that express something that you intend to be captured with this code – or to *not* be captured with this code. These examples are a great starting point for your list of examples of data fragments that should or should not be coded with your code.

Specifically, for every code, try to include at least one example in each of the four categories formed by crossing hits and misses with core and edge cases. Hits refer to examples of data fragments that *should* be coded with a code, whereas misses refer to examples of data fragments that *should not* be coded with a code. Core cases refer to examples of data fragments that, when people read your code label and description are *straightforward* to correctly classify as a hit or a miss. Edge cases refer to examples of data fragments that are more *ambiguous* and would be hard to correctly classify without the elaboration you added to your coding instruction when you worked on edge cases.

These two characteristics (hit vs. miss; and core vs. edge) combine to form the following four categories of examples:

- Core hits: Examples of data fragments that should clearly be coded with this code, even if somebody just reads the code label and description (let alone with the coding instruction).
- Core misses: Examples of data fragments that could be considered to belong to the code if somebody only reads the code label (and maybe still when they read the code description), but where the coding instruction clearly clarifies that it should *not* be coded with this code.
- Edge hits: Examples of data fragments that are manifestations of edge cases that fall within the scope of this code. As edge cases, whether these fragments should be coded with this code remains relatively ambiguous even with the coding instruction, but your consideration and explicit discussion of edge cases should help coders correctly classify this data fragment.
- Edge misses: Examples of data fragments that are manifestations of edge cases that fall without the scope of this code. As edge cases, whether these fragments should be coded with this code remains relatively ambiguous even with the coding instruction, but your consideration and explicit discussion of edge cases should help coders correctly classify this data fragment.

If you find that drafting this list of examples is hard, that could be an indication that your codes aren't sufficiently elaborated. If codes are still vaguely defined, it can be very hard to think of how they could look if they show up in qualitative data. Having relatively indeterminate codes does not need to be a problem: this mostly depends on your intention during the coding phase. If you're mostly towards the deductive end of the deductive-inductive spectrum, it is important that your codes are well-defined, unambiguous and unequivocal. However, if you intend to engage in mostly inductive coding, having codes that aren't comprehensively fleshed out is less of a problem. In that case, you're developing your initial code set as a tool to exercise transparency as to your initial expectations and ideas. If those are relatively vague, it is fine to have this reflected in relatively vague codes.

Because in some situations, it can be acceptable to have relatively underspecified codes, in such situations you can also decide to not include examples at all. However, do not make that decision too lightly: thinking about examples of the kinds of data fragments that you expect to find and how you would categorize them conceptually is a very useful exercise even if you aim to engage in mostly inductive coding. As discussed in the beginning of this chapter, it is easy to fool yourself, and though thinking about your expectations and how you see the world can be very hard, it is also a powerful tool to avoid fooling yourself – and others.

### 5.3.3 Piloting the coding instructions

Depending on how you plan to code in terms of the deductive-inductive coding spectrum, you may want to run a pilot with a small portion of the data. This will allow you to refine the coding instructions and the definitions. Because piloting effectively shifts a number of decisions from the coding stage to the preparation stage, piloting makes more sense as your aim is to code more deductively.

During piloting, you will encounter ambiguities in your coding instructions and corresponding code descriptions and will be able to resolve them. This enhances the consistency with which codes will be attaches to data fragments during the actual coding phase.

However, because process entails interaction with the data and can result in refining of your code structure, it can justifiably be seen as an activity that *belongs* in the coding phase. Especially if you take a more inductive approach, therefore, you may prefer to not conduct pilots.

## 5.4 Segmentation

In some projects, it can be useful to segment data. In fact, working with the ROCK requires at least one level of segmentation: segmentation of the data into utterances, as smallest codeable data

fragments. However, in many projects, utterances serve a purely pragmatic function, providing an anchor to attach codes to. Higher-level segmentation is meant to distinguish segments based on some definition. For example, you could segment in such a way that the utterances comprising participants' answers to different questions in an interview are in the same segment; or in such a way that when the data covers a different topic, a new segment starts. We will first discuss utterances as lowest-level segmentation, and then go into the higher-level segments.

### 5.4.1 Utterances

In any ROCK project, you need to decide what constitutes an utterance. Depending on your plans for your analysis, the impact of this decision on your results can be anything from trivial to decisive.

For example, if you aim to use a hierarchical organizational mode to describe people's thoughts and feelings related to, for example, how a global pandemic affected their social lives, it won't make a difference whether you define utterances as words, phrases, sentences, or paragraphs. In this approach, coding serves to organize the data, making it easy to view all data fragments coded with a specific code (or an ancestor of that code). If you also code inductively, in addition, your ultimate code structure will be a concise description of the patterns you found in the data.

In this scenario, to which word, sentence, or paragraph exactly you attach a code doesn't really matter. One of the main strengths of qualitative data is its richness, and this also means that interpreting data fragments usually requires their (rich) context. Therefore, when inspecting coded fragments, you usually include a certain number of preceding and following utterances to enable accurate interpretation. Therefore, you could say that you generally code a cluster of utterances: to which one specifically you attach a code doesn't really matter, because as long as you attach the code to one of them, you achieve your goals (i.e., selectively looking at the data fragments where a code appears, and optionally yielding an ultimate code structure that represents your results).

However, sometimes you have different aims. For example, you might have an epistemological, theoretical, and methodological framework that affords drawing inferences from code co-occurrences. That framework then dictates the conditions within which such inferences are valid. If you think that co-occurrence of two codes is indicative of, for example, the proximity of the concepts those codes represent in people's mental models of the world, for that inference to be valid the codes have to be occur sufficiently close together (what exactly "sufficiently close" is should be defined in the framework that affords the inferences in the first place).

In such situations, it can matter a great deal how utterances are defined. If co-occurrence is determined at the utterance level (instead of at a higher level of segmentation), then how much data constitutes an utterance determines how many co-occurrences will be found. In such situations, it is also important that coding is accurate: whereas in the first scenario, it doesn't matter much whether a code is attached to one utterance or the next (given that patterns will often manifest in the data diffusely, not cleanly organized within utterances) in the second scenario, the utterance a code is attached to determines which co-occurrences are produced.

You will generally find yourself in one of two situations. In the first scenario, as illustrated by the first example, there's no risk that your utterance definition biases your results. In that case using a sentences as an utterance seems a nice compromise between accuracy and feasibility. In the second scenario, as illustrated by the second example, there is a considerable risk that your utterance definition biases your results: for example, if your utterances are too large, inferences from co-occurrences are no longer valid; and if they are too small, you miss relevant co-occurrences. In that scenario, however, the very framework that affords drawing inferences from co-occurrences in the first place does so by virtue of a justification grounded in your epistemological, theoretical, and methodological assumptions, which therefore also prescribe the appropriate utterance definition(s).

### 5.4.2 Higher-level segmentation

In addition to the definition of utterances, sometimes you want to use higher-level segmentation. There are two common scenarios where you want to use higher-level segmentation. The first is pragmatic: you may want to distinguish different phases of data collection, or different stages in a process. The second is methodological: you may require higher-level segmentation to draw certain inferences. Another example of that same situation is when you are interested in code frequencies per segment, or when you want to map networked codes separately for each segment.

Higher-level segmentation is indicated in your sources through coding (but by adding section breaks instead of codes). That means that each type of higher-level segmentation also has a definition and coding instructions. Like for codes, even if you do not document those in advance, ultimately the segmentation will be applied accordingly to some system – documenting it makes it transparent, but doesn't create it.

Where definitions and coding instructions for codes capture the essence of the corresponding construct, definitions and coding instructions for segmentation instead capture transitions between segments. For example, if you want segments to represent different questions asked by an interviewer, the coding instruction might instruct coders to look for those questions in the interview transcript and insert section breaks just above each question.

Alternatively, you may want segments to represent turns of talk. In that case, your coding instructions would describe how you define a turn of talk. For example, if one person is speaking, but another briefly interjects to voice agreement, do you consider that a turn of talk? Or do turns of talk require something more, for example that the new speaker expresses something original? Or uses at least a clause? Or do you use another definition?

You may also want something (even) more complicated: for example, you may want segments to distinguish discussion of different topics. In that case you will have to define what a topic change is and how it can be recognized in a source with qualitative data. What exactly you end up choosing depends, again, on the implications of that choice, informed by your epistemological, theoretical, and methodological assumptions. Because these same assumptions inform your analyses and justify the inferences from the results of those analyses, they can also guide how you define topic changes.

## 5.5 Your initial code book

At this point, you should have a first version of your initial code book. It represents a manifestation of your initial expectations, assumptions, and ideas about what you will encounter in the data. It should normally consist of the following:

- A decision for a specific organizational mode, ideally with an explicit justification to help other researchers as well as future you understand why you chose that organizational mode;
- For each code:
  - A unique identifier for the code. Identifiers are machine-readable, always start with a letter, and can only contain a-z, A-Z, 0-9, and underscores (\_).
  - A human-readable label for the code. This is usually how you will refer to it in your manuscript and other presentations of your project.
  - A description of the code where you describe what it is intended to capture more in detail.
  - A coding instruction that can be applied to data fragments to arrive at a decision as to whether that code should be applied to that data fragment or not, ideally including explicit discussion of edge cases and cross-references to other codes where applicable.
  - Ideally, as a part of the coding instruction, examples from the four categories (core hits, core misses, edge hits, and edge misses).
- For each segmentation, the same as for every code.



You can include this initial code book when you preregister your study. Publicly freezing it like this is an efficient way to help yourself to not only not fool yourself, but also not fool others. It makes it much easier to compare the code book you end up with after you analyzed all the data with what you started out with in the first place. In addition, it helps you to recognize when you shift in your understanding based on your interaction with the data by making changes from your initial expectations more salient, creating clear opportunities for you to justify those decisions.

It is important to realize that like a preregistration, your initial code book is a plan, not a prison: you publicly freeze it to be transparent, not to create a straightjacket to dogmatically adhere to. Even when your goal is to work completely deductively, your code book may evolve over the course of your coding. That is one of the reasons why preregistration is so important: it helps keep track of such changes and comprehensively document their justifications.



## Chapter 6

# Planning and Conducting Interviews

### 6.1 Planning

For more information about data sharing, see the Psy Ops Guide.

For more information about informed consents, as well as some example templates, see the Psy Ops Guide.



# Chapter 7

## Coding

Coding is the process of attaching codes to data fragments.

### 7.1 Coding

### 7.2 Segmenting

Segmenting is the process of delineating data into segments. Segments can have different definitions, but once defined and applied, each segment binds together data fragments that exhibit some coherence in terms of the corresponding segment definition.

A relatively simple segment definition is a sentence. For sources that are interview transcripts, another example is an interviewee's response to each question.



## Chapter 8

# Preregistering qualitative research

This chapter will cover items that are commonly included in (pre)registration forms for qualitative research. (Pre)registration is the process of freezing your project and relevant documentation about the project before, during, and/or after the project.

Open Science frameworks, such as the Open Science Framework (<https://osf.io>), generally freeze all files in your project repository if you register your project. This preserves that state of your project, along with what you entered in the (pre)registration form, for future reference. In a way, you're facilitating playing archeologist with your project.

### 8.1 Types of registration

#### 8.1.1 Preregistration

The most common type of registrations is preregistration, where you freeze your plans and expectations beforehand. This has a number of benefits. One is that it's very helpful to avoid fooling yourself later on. For example, the process of collecting and interacting with the data shapes and changes how you see (part of) the world. Because human memory is pretty fallible and a far cry from a video or a computer's hard disk (which usually allow perfect recollection of stored data), these changes in how you see the world bias your memories.

It is easy to erroneously be convinced that you already expected some findings - or conversely (and perhaps worse), that patterns you observed in the data represent new findings, rather than a reflection of your prior expectations. In qualitative research (and, by the way in quantitative research), the choices you make during (the design of) your data collection and (the design of) your analyses are influenced in part by your expectations, being up-front about them before your data collection starts is very helpful to get a grip on that influence.

#### 8.1.2 Repeated registration

Because expectations and insights change, and often, during a project, it is necessary or beneficial to change your approach, it can be helpful to repeatedly register your project. A straightforward example is to freeze a registration when you submit your request for ethical approval; and then freeze it again when you made changes in response to comments or questions from the ethical committee; and then freeze it again just before you start data collection.

Another example is freezing a registration halfway during your data collection so that you can document changes in data collection procedures (e.g. you might add questions to the interview scheme). Another

example is freezing a registration when you submit a manuscript to a journal; and another when you resubmit after having revised the manuscript, potentially having done additional or different analyses.

This process allows you to keep track of your changing expectations, representations, and analytical choices, as well as facilitating insight into such changes over multiple projects.

## 8.2 Common registration form items

### 8.2.1 Prior expectations and/or initial code structure

Qualitative data is usually coded by humans because as yet, computers are unable to effectively process language and extract meaning. Language is messy and often ambiguous, and context is often vital for correct interpretation, where the amount of context taken into account can change the meaning of an expression. Humans are quite good at this, partly because our psychological machinery is very good at pattern recognition and categorization. This skill also comes at a price, though: humans categorize too enthusiastically, often seeing patterns where none exist. Using human coders is therefore both inevitable and risky.

Before coders start coding data, they often have expectations and mental representations of the bits of the world that are captured in the data. For example, coders will often be aware of the research questions in the project. Those research questions did not come about in a vacuum, but have scientific, cultural, societal, political and subjective provenance and context. Similarly, the coders are human, and as such, have representations of the world that often manifest as predispositions to categorize parts of the world in a certain manner. For researchers, those predispositions will often be shaped by the theories that they are familiar with, even if those theories are not focal in the study they code data for. These predispositions will shape how the coders will code the data: they are the lens through which coders will code the data.

Not all humans excel in metacognitive skills such as introspection, and therefore these predispositions can be hard to identify and accurately describe in a (pre)registration. One approach to facilitate this process is to use an initial coding structure with clear coding instructions accompanying each code. The advantage of this approach is that it provides a vehicle for explicitly discussing the framework that will be used for coding, simultaneously aligning coder's lenses and making the resulting 'shared lens' explicit.

However, not all projects lend themselves to development of such an initial coding structure. It is possible that there are no expectations at all. This is exceedingly rare – given how hard metacognition can be, and given that humans automatically categorize the world, if researchers think they have no prior expectations it may well indicate a lack of awareness of their implicit expectations – but it can happen, for example if the coders are not researchers, have no theoretical knowledge or naive mental models of the substance to be coded, and are unaware of the goals of the study and the research questions.

A more common scenario is that the project was designed to use multiple coders that will code inductively. This can be useful to explore variation in the coding structures that are developed by the independent coders, and benefit from the discussions that serve to reconcile those differences into one unifying coding structure (that is then often better understood and justified, provided the discussions are comprehensively documented).

In that scenario, each coder will however still have prior expectations and categorization tendencies that will shape how they will code the data. It is likely these will manifest in the coding structure each coder produced. The patterns in those coding structures can then easily be misunderstood as conveying characteristics of the data, whereas in fact, they represent expectations that existed prior to, and independent of, the data collection and coding. If the coders share the same background, this is even more risky: the convergences in their coding structures can easily be misinterpreted as evidence of the 'truthfulness' of the patterns represented by those parts of the coding structure.



This risk of bias can be ameliorated by having each coder make their expectations, and if possible, their initial coding structure, explicit individually and independently. Ideally, they include coding instructions, just as you would for an initial coding tree if you engage in deductive coding to enable replication by other coders.[^One could argue that, given the inevitability of pre-existing expectations and categorization tendencies, coding by humans is never truly inductive – and therefore, acknowledging the deductive aspects is a powerful tool to avoid fooling yourself during analysis and interpretation of the data.] This approach requires one of the project member who will not themselves engage in coding to compile these initial expectations or coding structures and add them into the preregistration form just before it is publicly frozen. This safeguards the independence of the coders (i.e. if these initial expectations or coding structures are shared between coders, they will no longer be independent).



## Chapter 9

# Reporting Qualitative Research

Best practices for reporting differ between different types of research. For example, in most quantitative research, the employed statistical models usually require that sampling procedures are designed to sample randomly from the population, and accurate estimation with those statistical models usually requires considerable sample sizes. Because the sampling and measurement error can then be modelled, uncertainty can be estimated, which then enables reporting quantitative results in a transparent and integreous manner. In most qualitative research, on the other hand, sampling procedures are designed to optimize sample heterogeneity, data collection can be partly driven by the observations and is therefore less systematic, and the error in sampling and observation cannot be modeled.

Fundamental differences such as these have a number of implications for how research is reported. In this Chapter, we will list a number of best practices for reporting qualitative research, paying special attention to similarities and differences between qualitative and quantitative methods. Consistent with the Justify Everything principle, we will also provide justifications for these best practices. We will follow the conventional manuscript structure of introduction, methods, results, and discussion.

### 9.1 Introduction

### 9.2 Methods

#### 9.2.1 Full Disclosure

This section includes the link to the repository containing the Full Disclosure package for this study. A Full Disclosure Package consists of a Replication Package and an Analysis Package.

The Replication Package commonly contains everything required for, or facilitating, replication of the study, such as the request for ethical approval and the confirmation letter granting ethical approval; the communication templates for communicating with participants; the recruitment protocols; the interview scheme or topic list; the protocols for support for participants and for interviewers, for studies where the interviews might touch upon sensitive topics; and the transcription procedures; the data management plans. The Replication Package should allow other researchers to replicate your data collection with minimal effort.

The Analysis Package commonly contains the (anonymized) raw data; the (anonymized) processed data; the documentation of the analysis steps and decisions taken; justifications of those decisions; and your results. The Analysis Package should allow other researchers to replicate your data analysis with minimal effort.

### 9.2.2 Materials

This section describes the used materials, such as the interview scheme or topic list; the communication templates for communicating with participants; the recruitment protocols; and the protocols for support for participants and for interviewers, for studies where the interviews might touch upon sensitive topics. In this section, also describe how data will be recorded.

### 9.2.3 Sampling strategy

In this section, describe the sample composition you aim to achieve, and how you operationalized these goals.

### 9.2.4 Analysis plan

Here, describe the analysis plan. This includes the type of coding (e.g. inductive, deductive, or a combination), how many coders will be involved and how the sources will be divided, etc etc etc

## 9.3 Results

### 9.3.1 Participants

In this section, describe the participants of the study. This ideally happens in a similar manner for quantitative and qualitative research. Where the Sampling Strategy section in the Methods section described the aims and strategies, this section describes the results of those efforts. Note that unless the study is a case study of one or more cases, the individuals are not the primary interest. Therefore, this section should normally describe your *sample*, and not be a list of descriptions of each participant. To the degree that specific participant characteristics are important for contextualizing specific source fragments as described later on, list those characteristics at that point in the narrative - do not burden readers with mixing and matching throughout the manuscript. Note that more extensive descriptions can always be include in the repository accompanying the manuscript.

### 9.3.2 Background

In some cases it may be beneficial to start with a description of the context of the participants and data collection efforts. Such contextualisation may be important to properly interpret the results, even though the information itself may not pertain to the relevant research questions. Therefore, it can be useful to separate this description from the primary results.

### 9.3.3 Results

Then, describe the results.

## 9.4 Discussion

## 9.5 Reporting standards

One guideline for reporting qualitative research is formed by the consolidated criteria for reporting qualitative research (COREQ). This is available at <https://academic.oup.com/intqhc/article/19/6/>

349/1791966.

The items are ...



## **Part II**

# **The ROCK**





# Chapter 10

## The ROCK vocabulary

In the plethora of qualitative approaches, many different terms exist and often partially overlap. The ROCK standard uses the terms listed below. Below this section, we included a list of definitions of ENA terms.

### 10.1 ROCK terms

**attribute** A property or characteristic of a *class instance* (e.g. a participant), for example demographic variables such as interviewee age, gender, education level. Attributes can also be characteristics of instances of classes that are not persons, such as interview venue (an attribute can be e.g. whether it was crowded or not) or interviewer (an attribute can be e.g. the interviewer’s age).

**case** A data provider, such as a participant in a study. Cases are a *class*, with every individual case being a *class instance*.

**class instance** A data provider, context, or other description of data collection. In interview studies, a class instance is usually a specific person. Assigning utterances to class instances is a means to efficiently associate *attributes* to many utterances in one go. Class instances can also be used to associate other information to many utterances, such as the interviewer, the place where an interview took place, or the time of day. Examples of class instances are “participant 4” to identify a person, “14:00” to identify the time of an interview, “meeting room B” to identify the location of an interview.

**class instance identifier** The unique identifier for a class instance. Examples of class instance identifiers in the ROCK are `[[cid:1]]`, `[[cid:participant_A]]`, `[[locationId:school]]`, and `[[timeId:morning]]`, where the first part denotes the class (e.g., `cid` stands for ‘case identifier’) and the second part identifies the instance of that class (e.g. 1 and `participant_A` can refer to participants in a study). Class instance identifiers can be ephemeral or persistent: if persistent, they will apply not only to the *utterance* where they occur, but also to all following utterances in the same *source* until a new class instance identifier for the same class is encountered.

**code** A symbol that represents data fragments that are somehow similar, that similarity being the essence of the relevant code. Such a code usually represents a concept. Codes can vary from simple descriptions, for example to denote that the coded fragment concerns a topic such as “leisure activities”, to complex constructs, for example to denote that the coded fragment likely expresses psychological aspects that fall within the definition of a construct called “perceived autonomy”. Codes are represented in a *source* using a *code identifier*. In addition to that (machine-readable) identifier, codes commonly have a (human-readable) short label, a longer description, and coding instructions, ideally including examples.

**comments** In the ROCK standard, all utterances starting with a hash, #, are considered comments and are ignored. Note that the hash may not be preceded with whitespace: it has to be the

first character in the *utterance* (usually, given the default *utterance marker*, this means the first character on each line).

**code delimiter** A combination of two text strings that enclose *code identifiers* in coded *sources* to indicate which *utterances* are coded with which *codes*. In the ROCK standard (and in the default **rock** R package settings), the code delimiters are pairs of square brackets: `[[` and `]]` (e.g. `[[code]]`).

**code identifier** A brief *identifier* to uniquely identify a *code*. Code identifiers are used to represent the corresponding *code* when coding *sources*: they are then enclosed with the *code delimiters* (`[[` and `]]` by default, e.g. `[[code]]`). Note the difference with *coder identifiers*, which represent coders instead of codes; and remember that like all identifiers, coder identifiers may only contain Latin letters, Arabic numerals, and underscores (and have to start with a letter).

**code structure** A set of *codes* in a given organizational mode. Three common organizational modes are a flat code structure (i.e. no structure), a hierarchy, and a network.

**code tree** If a hierarchical organizational mode is used, the code tree represents the hierarchy of codes used to *code* one or more *sources*.

**code value** A code value is a way to efficiently attach values to codes. They consist of a (potentially *hierarchically marked*) *code identifier* immediately followed by two pipe characters (`||`) and the relevant value, delimited with the *code delimiters*. For example, code values can be used to code the intensity of a statement using `[[intensity||1]]` or `[[intensity||low]]`, or the valence of a statement using `[[valence||positive]]` or `[[valence||neutral]]`. When parsing the coded sources into the qualitative data table, where regular codes yield 0s and 1s denoting whether a given utterance was coded with that code code, code values yield the specified value in the corresponding column. Note that this functionality is somewhat similar to using *attributes* to efficiently attach information to *utterances*. The difference is that attributes are applied to all utterances coded with the corresponding *class instance identifier*, and the most commonly used class instance identifiers (case identifiers, coder identifiers, and item identifiers) are set as *persistent identifiers*, which means they normally are automatically applied to many utterances. Code values, on the other hand, can be used to attach values to single utterances.

**coder identifier** A unique identifier for each coder. These are typically used when using multiple independent coders. By default, when used in a source, they are delimited using the *code delimiters* (`[[` and `]]` in the ROCK standard) and are preceded by `coderId=`. For example, the coder identifier “`coder_1`” would be used in a source as `[[coderId=coder_1]]`. Note the difference with *code identifiers*, which represent codes instead of coders; and remember that like all identifiers, code identifiers may only contain Latin letters, Arabic numerals, and underscores (and have to start with a letter).

**fragment** A part of a *source* (one or more consecutive characters, such as one or more words, sentences, or paragraphs). A data fragment can be considered somewhat akin to a ‘data point’ in a tabular data set, except that data fragments have not set size, and in that sense are recursive (i.e. data fragments contain other data fragments, e.g., paragraphs contain sentences). Data fragments are useful to refer to parts of a qualitative dataset completely independent of any segmentation that may have been applied.

**hierarchy marker** A symbol that represents hierarchical levels and is used to separate *code identifiers* when they are added to a *source*. In the ROCK standard (and the default R **rock** package settings), the hierarchy marker is the greater-than symbol, `>` (e.g. `[[parentCode>childCode]]`).

**identifier** A unique character sequence that uniquely identifies something. Identifiers always have to start with a lowercase or uppercase Latin letter (a-z or A-Z) and may only contain Latin letters, Arabic numerals (0-9), and underscores (`_`). The **rock** R package will often use identifiers as variable names: for example, code identifiers become variable names in the qualitative data table. Examples of well-known identifiers are Uniform Resource Locators (URLs, commonly used for websites); Digital Object Identifiers (DOIs, commonly used for scientific articles); and the International Standard Book Number (ISBN, commonly used for books). The ROCK implements a way to generate and specify identifiers for *utterances* and a way to add other identifiers to a *source*, such as for *codes* and for *class instances*. In the ROCK, when the term “identifier” is used without specifying what type of identifier is meant, usually *class instance identifiers* are

meant, such as identifiers for cases, coders, or items.

**nesting marker** A symbol that represents nesting (or threading) in a *source*. In the ROCK standard (and the default R **rock** package settings), the nesting marker is the tilde, ~, and has to be placed as the first character in an utterance (disregarding *whitespace* and the *utterance identifier*). For example, to denote that an utterance is a response to the preceding utterance, start the second utterance with a tilde (e.g. `[[uid:xxx]] ~ This is a response`). If the following utterance also starts with a tilde, it is considered to also be a response to the first utterance; if it starts with two tildes, the nesting deepens and that third utterance is considered to be a response to the second utterance (e.g. `[[uid:xxx]] ~~ This is a response to the response`).

**persistent identifiers** Whereas regular identifiers are only applied to the utterance where they occur, persistent identifiers are automatically applied to all following utterances until the end of the source or until a persistent identifier for another instance of the same class is encountered. For example, by default, case identifiers, coders identifiers, and item identifiers are configured to be persistent. This means that if the case identifier for a given participant is applied to an utterance, all subsequent utterances will be considered to belong to that same participant until another case identifier is encountered.

**section** A delimited *fragment* of a source, also called segment or stanza.

**section break** A break between two sections. In other words, section breaks split up *sources* into *sections*. The ROCK standard allows parallel use of multiple types of section breaks: for example, one type of section break can indicate paragraph breaks, whereas another type of section break can indicate where an interviewer asks a new question, and yet another type can indicate where there is a turn of talk between participants in a discussion. In the R **rock** package, by default anything matching the regular expression `---<<[a-zA-Z0-9_]+>>---` is considered a section break, where the sequence in between the smaller than signs and the greater than signs means “one or more of a Latin letter, an Arabic numeral, and an underscore” (i.e. the same pattern that holds for *identifiers*).

**section break identifier** A sequence of characters that represents a *section* break.

**segment** See *section*.

**source** A plain text file that describes or captures a bit of reality. The most common sources in research with humans (e.g. anthropology and psychology) are interview transcripts, but sources can also be internet content, archive materials, meeting minutes, descriptions of photographs, or timestamped descriptions of video material. Note that a source does not necessarily correspond one-on-one to any sensible delineation of reality. Often, each separate source will hold the data from one interview, but this is not necessary. Many interviews can also be combined in one source, and one interview can also be distributed over many sources. To distinguish, for example, interviews, use *class instances*. A source is simply the text file holding one or more utterances.

**utterance** The shortest *codable fragment* of a *source*. They are separated by *utterance markers*. In the ROCK standard (and the default R **rock** package settings), these are line breaks (“\n”), which means that each utterance is on a line of its own in a source. Utterances will often (but not necessarily) correspond to sentences. Other examples of utterances are words, clauses, phrases, or other constituents, paragraphs, or social media posts.

**utterance marker** The sequence of characters that delimits *utterances*. In the ROCK standard (and the default R **rock** package settings), these are line breaks (“\n”).

**utterance identifier** A unique identifier for an utterance. These are used to match utterances when multiple independent coders are used. By default, when used in a source, they are delimited using the *code delimiters* ([`[` and `]`] in the ROCK standard) and are preceded by `uid=`. For example, the utterance identifier (“UID”) “7fgglz2n” would be used in a source as `[[uid=7fgglz2n]]`. The R **rock** package has functions to automatically attach UIDs to sources. UIDs are normally placed at the beginning of each utterance.

**whitespace** Invisible characters such as spaces and tabs.

**YAML** YAML is a standard for encoding data in plain text files in a way that is easily readable by humans. The ROCK standard uses the YAML standard for specifications of *attributes* as well as deductive code structures. YAML is a recursive acronym that stands for “YAML Ain’t Markup Language”, and is technically a JSON (Javascript Serial Object Notation) superset, which means

that all JSON is valid YAML.

## 10.2 ENA terms

**code** A construct of interest. Codes are used to label qualitative data.

**conversation** A set of one or more stanzas. Code co-occurrences are aggregated on this level of segmentation.

**co-occurrence** The occurrence of two or more codes within a designated segment of data (stanza window).

**edge** In an ENA network, an edge represents *normalized co-occurrence of codes*.

**node** In an ENA network, a node represents a code.

**normalization** Normalization refers to the division of a vector of co-occurrences by the number of utterances ... ? Or does it?

**stanza** A set of one or more utterances. Code co-occurrences are computed on this level of segmentation.

**stanza window** A specific operationalization of stanza (e.g., moving stanza window, infinite stanza window, whole conversation).

**unit** All lines (utterances) that are included in an epistemic network model.

**utterance** A line of data; the smallest codable segment of data.

**vector** A vector is a sequence of one or more data points (e.g. numbers), for example a series of 0s and 1s.

# Chapter 11

## The ROCK standard

The ROCK standard has been developed as an open standard for qualitative data analysis. It follows the principles that also guided the development of the Markdown and YAML standards: prioritizing human-readability while retaining machine-readability. The aim of the ROCK is to provide a standard that enables researchers to exchange data and analyses in a format that is readable even without running any specific software. In other words, coded transcripts should be readable as is.

This open standard enables development of programs or scripts to perform specific functions that are not yet present in any of the existing applications that support the ROCK format. In addition, this enables all existing qualitative data analysis programs to import data files in this format and to export to this format.

In this chapter, the vocabulary explained in Chapter 10 is used to describe the ROCK standard. Qualitative data files that implement the ROCK standard can be recognized by their extension: `.rock`. These files normally follow the conventions set out in this chapter.

### 11.0.1 Sources, utterances, and sections

*Sources* are plain-text files that contain qualitative data. These data are segmented into the smallest codable unit which is called an *utterance*. In the ROCK standard, utterances are separated by a newline character (“`\n`”), which means that every line in a source is an utterance. Often, utterances will be sentences, but not necessarily.

A source will often, but not necessarily, contain data that is somehow related. For example, different interview transcripts may be stored in different sources (i.e. different plain-text files). It is also possible to split up data over several sources, or combine data from multiple data collections in one source. A source can be seen as a logistical necessity, but sources do not have meaning: in the ROCK standard, whether two utterances are in the same source or not is not relevant.

There are two approaches to represent meaningful grouping of utterances. One is using persistent class instance identifiers, which will be discussed below. The other is using *sections*: sections segment the data. Section breaks occur between two utterances, separated from those utterances by newline characters. They start with three dashes and a smaller-than sign, followed by an identifier for the section break, and end with a greater-than sign and three dashes. For example, valid section breaks are `---<turn_of_talk>---` and `---<new_question>---`.

### 11.0.2 Codes

Codes are identified by code identifiers. Code identifiers are unique strings of characters (specifically, lower or uppercase Latin letters, Arabic numerals, and underscores) to represent a code in sources.

These identifiers are placed in between two pairs of square brackets ([[ and ]]). Codes are designated per utterance, or in other words, per line. As many codes can be specified per line as one wishes. For example, see these two lines (utterances):

```
So what went right [[reflection_positive]]
What went wrong [[reflection_negative]]
```

The first line is coded with `reflection_positive`, and the second line with code `reflection_negative`.

### 11.0.3 Structuring inductive codes

When engaging in inductive coding (i.e. when not working with a prespecified code structure, but instead developing the code structure as one goes along; see the section below re: deductive coding), to represent the hierarchical structure of the codes, a greater-than sign (“>”) can be used. For example, perhaps a researcher wants to specify a parent code such as `reflection` with two child codes such as `positive` and `negative`. This helps one to identify patterns in the data, and makes it possible to easily extract all utterances coded as any type of reflection. For example, see the same fragment but coded in two levels:

```
So what went right [[reflection>positive]]
What went wrong [[reflection>negative]]
```

When this source is parsed by `rock`, it will recognize these codes and their structure, and it will generate the corresponding hierarchical coding structure, as illustrated in the more extensive example below.

### 11.0.4 Class instance identifiers

It is often desirable to attach specific attributes to utterances. For example, one may want to compare the patterns in codes between different categories of participants, such as those who do and do not own a car, or those that listen to progressive metal versus those that listen to psychedelic trance. Instead of coding all utterances with all relevant attributes, instead, it is possible to specify class instance identifiers to easily link utterances to characteristics of the data provision (such as data providers, for example participants, or the moment of data collection, for example daytime or nighttime, or winter or summer, or the location of data collection, such as in a busy place or in a silent office).

By default, three types of class instance identifiers are specified: case identifiers, coder identifiers, and item identifiers. They are again specified using two pairs of square brackets, but this time, the opening brackets are immediately followed by a string of identifying characters (the class instance identifier), followed by an equals sign, and then by the unique identifier. This may seem a bit abstract; it will become clearer as we look at the first example.

#### 11.0.4.1 Case identifiers

Case identifiers can be used to link utterances to data providers, such as participants. Their class instance identifier is `cid`, and by default, their full regular expression is `\[[cid=([a-zA-Z0-9_]+)\]`. A source excerpt coded with only case identifiers may look like this:

```
CAIAPHAS: No, wait! We need a more permanent solution to our problem. [[cid=1]]
```

```
ANNAS: What then to do about Jesus of Nazareth? Miracle wonderman, hero of fools. [[cid=2]]
```

PRIEST THREE: No riots, no army, no fighting, no slogans. [[cid=3]]

CAIAPHAS: One thing I'll say for him -- Jesus is cool. [[cid=1]]

ANNAS: We dare not leave him to his own devices. His half-witted fans will get out of control. [[cid=

(Note that in this example, the names of the participants were retained; normally, the researcher would anonymize the transcripts so as to allow publication of the coded transcripts.)

When **rock** parses this source, it will know that the first and fourth utterances belong to the same case, as do the second and fifth. The attributes specified for these cases will then be attached to these utterances (see the section about attributes below).

Class instance identifiers have a shorthand alias, which is used in the codes themselves (in this example, **cid**), and a longer version, which for case identifiers, is **caseId**. This longer version is used when specifying the attributes (see the section below).

#### 11.0.4.2 Stanza identifiers

A stanza is a unit of analysis in ENA analysis (see the glossary for the exact definition).

### 11.0.5 Specifying deductive coding structures

When a researcher works with a prespecified coding structure (i.e. engages in deductive coding), they only use codes that were determined a priori. Like in inductive coding, there are often multiple levels in such a coding structure, with the codes organised hierarchically. To efficiently be able to collapse codes to higher levels, **rock** needs to know the deductive coding structure. This can be specified using YAML fragments in the sources. YAML fragments are, by default, delimited by two lines that each contain only three dashes (---). Between those delimiters, YAML (a recursive acronym that stands for ‘YAML ain’t markup language’) can be specified. Specifically, in YAML terminology, each fragment should be a sequence of mappings that is named **codes**.

The code tree specified in the section on inductive coding, for example, can be efficiently specified as a deductive coding structure like this:

```
---
codes:
  -
    id: reflection
    children:
      -
        id: positive
      -
        id: negative
---
```

If all children of a code are so-called ‘leaves’ (i.e. in the code tree, they have no children of their own<sup>^</sup>) they can be specified more efficiently:

```
---
codes:
  -
```

```

    id: reflection
    children: ["positive", "negative"]
---
```

When `rock` parses the sources, it will collect all such code specifications and combined them into one coding three using each code's identifiers. It is possible to specify a parent in other code specification fragment by adding the field `parentId`. For example, in another source, we could add this fragment:

```

---
codes:
-
    id: neutral
    parentId: reflection
---
```

This would add `neutral` as a sibling to `positive` and `negative`.

### 11.0.6 Specifying attributes

Attributes are also specified using YAML fragments in one or more sources. These fragments have to start with `ROCK_attributes`, and have to contain the long version of the class instance identifiers. By default, the long version of the case identifier is “`caseId`” (the shorthand alias, used when coding, is “`cid`”). Each of the attributes that is specified will appear in the qualitative data table as a column.

```

---
ROCK_attributes:
-
    caseId: 1
    hair_color: grey
    age: 50
-
    caseId: 2
    hair_color: brown
    age: 40
-
    caseId: 3
    hair_color: red
    age: 45
---
```

## 11.1 Examples

### 11.1.1 Section breaks

```

So what went right
What went wrong
---paragraph-break---
Was it a story
or was it a song
---paragraph-break---
Was it over night
```



Or did it take you long  
 ---paragraph-break---  
 Was knowing your weakness  
 what made you strong

*Source excerpt as example of section breaks (lyrics from Smiley Faces by Gnarlz Barclay)*

### 11.1.2 Identifiers

CAIAPHAS

No, wait! We need a more permanent solution to our problem.

ANNAS

What then to do about Jesus of Nazareth? Miracle wonderman, hero of fools.

PRIEST THREE

No riots, no army, no fighting, no slogans.

CAIAPHAS

One thing I'll say for him -- Jesus is cool.

ANNAS

We dare not leave him to his own devices. His half-witted fans will get out of control.

PRIESTS

But how can we stop him? His glamour increases by leaps every moment; he's top of the poll.

CAIAPHAS

I see bad things arising. The crowd crown him king; which the Romans would ban.

I see blood and destruction, Our elimination because of one man. Blood and destruction because of

ALL (inside)

Because, because, because of one man.

CAIAPHAS

Our elimination because of one man.

ALL (inside)

Because, because, because of one, 'cause of one, 'cause of one man.

PRIEST THREE

What then to do about this Jesus-mania?

ANNAS

How do we deal with a carpenter king?

PRIESTS

Where do we start with a man who is bigger Than John was when John did his baptism thing?

CAIAPHAS

Fools, you have no perception! The stakes we are gambling are frighteningly high!

We must crush him completely, So like John before him, this Jesus must die. For the sake of the n

*This Jesus Must Die by Andrew Lloyd Webber*

### 11.1.3 Codes

## Chapter 12

# The rock R package

The **rock** R package implements the ROCK standard for qualitative data analysis. It is an extension to R, a program that was originally a statistical programming language. R is not only open source, but also has a flexible infrastructure allowing easy extension with user-contributed packages. Therefore, R is quickly becoming a multipurpose scientific toolkit, and one of its tools is the **rock** package.

When using R, most people use RStudio, a so-called integrated development environment. It has many features that make using R much more userfriendly and efficient. In this book, where we refer to using R, we actually mean using R through RStudio. Both R and RStudio are Free/Libre Open Source Software (FLOSS) solutions. This means that they are free to download and install in perpetuity.

### 12.1 Downloading and installing R and RStudio

Because RStudio makes using R considerably more userfriendly (and pretty), in this book, we will always use R through RStudio. Therefore, throughout this book, when we refer to R, we actually mean using R through RStudio.

R can be downloaded from <https://cloud.r-project.org/>:<sup>1</sup> click the “Download R for ...” link that matches your operating system, and follow the instructions to download the right version. You don’t have to start R - it just needs to be installed on your system. RStudio will normally find it on its own.

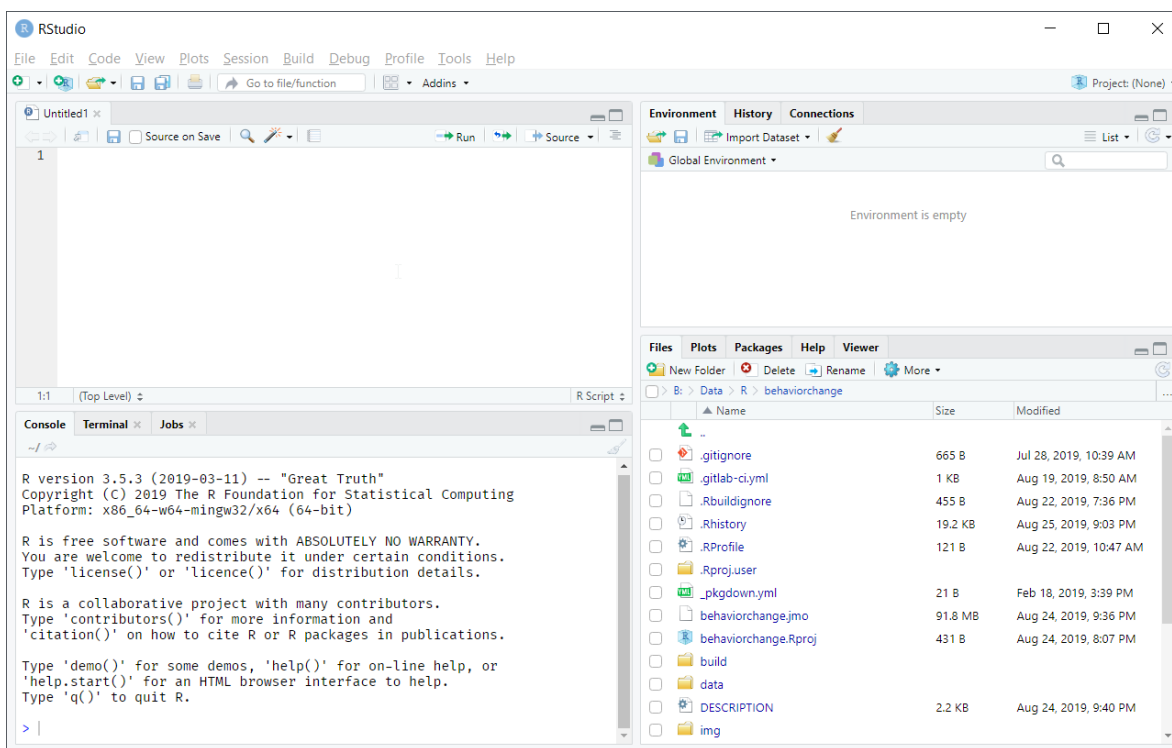
RStudio can be downloaded from <https://www.rstudio.com/products/rstudio/download/>. Once it is installed, you can start it, in which case you should see something similar to what is shown in Figure 12.1.<sup>2</sup>

R itself lives in the bottom-left pane, the console. Here, you can interact directly with R. You can open R scripts in the top-left pane: these are text files with the commands you want R to execute. The top-right pane contains the Environment tab, which shows all loaded datasets and variables; the History tab, which shows the commands you used; and the Connections and Build tabs, which you will not need. The bottom-right pane contains a Files tab, showing files on your computer; a Plots tab, which shows plots you created; a Packages tab, which shows the packages you have installed; a Help tab, which shows help pages about specific functions; and a Viewer tab, which can show HTML content that was generated in R.

---

<sup>1</sup>Yes, that page looks a bit outdated.

<sup>2</sup>It is easy to change RStudio’s appearance; simply open the options dialog by opening the Tools menu and then selecting the Global Options; in section Appearance, the theme can be selected.



**Figure 12.1:** The RStudio integrated development interface (IDE).

## 12.2 Downloading and installing the rock package

The **rock** package can be installed by going to the console (bottom-left tab) and typing:

```
install.packages("rock");
```

This will connect to the Comprehensive R Archive Network (CRAN) and download and install the **rock** package. If you feel adventurous, you can instead install the one of the two development versions. One is the most current production version, and the other is the development ('dev') version. The most current production version will generally be as stable as versions on CRAN, and will contain more features. This version will contain all features discussed in this book. The dev version contains work on new features. This also means, however, that it may contain bugs.

To conveniently install the most recent production and dev versions, another package exists called **remotes**. You can install this using this command:

```
install.packages("remotes");
```

Then, to install the most up-to-date production version, use:

```
remotes::install_gitlab("r-packages/rock");
```

And to install the current dev version, use:

```
remotes::install_gitlab("r-packages/rock@dev");
```

More information about the `rock` package can be found at its so-called pkgdown website, which is located at <http://r-packages.gitlab.io/rock>.

## 12.3 rock functions

### 12.3.1 `clean_source` and `clean_sources`

Sometimes, sources are a bit messy.<sup>3</sup> In such cases, it can be efficient to preprocess them and perform some search and replace actions. This can be done for one or multiple source files using `clean_source` (for one file) and `clean_sources` (for multiple files; it basically just calls `clean_transcript` for multiple files).

For example, a researcher will often want every sentence, as transcribed, to be on its own line (as lines correspond to utterances). In fact, this is the basic function of the `clean_source` function: by default, if used without other arguments, they try to (more or less smartly) split a transcript such that each transcribed sentence (as marked by a period (.), a question mark (?), an exclamation mark (!), or an ellipsis (...)) ends up on its own line. Before doing this, `clean_source` replaces all occurrences of exactly consecutive periods (..) with one period, all occurrences of four or more consecutive periods with three periods, and all occurrences of three or more newlines (\n) with two newlines.

But this function can also be used to perform additional (or other) replacements. For example, imagine that a transcriber used a dash at the beginning of a line, followed by a space, to indicate when a person starts talking, like this:

- Something said by one speaker
- Something said by another speaker

To easily group all utterances by the same person together, it would be convenient if this was expressed in the source file in a way that fits with ROCK's conventions. That sequence of characters (actually a newline character (\n) followed by a dash (-) followed by a whitespace character (\s)) can be converted into section break '---turn-of-talk---' with this command:

```
rock::clean_source(
  input = "
- Something said by one speaker
- Something said by another speaker
",
  replacementsPre = list(c("\\n\\s", "\\n---turn-of-talk---\\n")));
```

This will change those that bit of transcript into:

```
---turn-of-talk---
Something said by one speaker
---turn-of-talk---
Something said by another speaker
```

---

<sup>3</sup>Well, they are messy more often than not, unfortunately.

(You can copy-paste the command above into R and test this, assuming you have the **rock** package installed. Note that by default, R doesn't print newline characters as newline characters. To show newline characters as newlines, wrap the command in the **cat** command.)

To also maintain the default replacements, more can be added by specifying them in argument **extraReplacements** instead of **replacementsPre** (or **replacementsPost**). For **clean\_source**, as the first argument (**input**), either a character vector (like in the example above) or a path to a file can be specified, in which case the files contents will be read. If the second argument (**outputFile**) is specified, the result is saved to that file; if not, it is returned (and printed by R).

### 12.3.1.1 A word of caution

If you use this function to clean one or more transcripts, make sure that whenever you edit the **outputFile**, you save it under another name! Otherwise, rerunning the script to clean the transcripts will overwrite your edits. By default, the **rock** option “**preventOverwriting**” is set to **TRUE**, so by default, if a file already exists on disk, it is never overwritten. You can change this behavior for one function by specifying **preventOverwriting=FALSE** as a function argument. You can also change this for all functions by changing the option, with the following command:

```
rock:::opts$set(preventOverwriting=FALSE);
```

## 12.4 rock options and defaults

Although the behavior of the **rock** functions is mostly controlled by specifying the relevant arguments, the **rock** package also has many options that you can just specify once and that will then be used by all **rock** functions. This saves you from repeating the same arguments every time you call the **rock** functions.

Almost all of these options have default settings, many of which implement the ROCK standard, but some relating to project-level settings. We'll first discuss the project-level options, and then the options that implement the ROCK standard.

### 12.4.1 Project-level settings

### 12.4.2 Options implementing the ROCK standard

**utteranceMarker** The utterance marker: the string of characters used to separate utterances (i.e. marking the end of one utterance and the beginning of the next one).

## Chapter 13

# Recoding with the rock R package

### 13.1 Initial coding versus recoding

We use recoding to refer to the coding decisions and actions taken after the initial coding. In initial coding, codes are often attached to utterances using a text editor (such as Notepad++) or a graphical user interface (such as iROCK). These decisions are often based on categorization of utterances, and sometimes initial coding mostly concerns indexing, where the data is organized superficially, for example based on topic, instead of looking for the patterns that the researcher is interested in.

If the aim is to work in a transparent, reproducible manner, initial coding will often occur based on instructions in a coding book; if not decisions during initial coding will be based on implicit definitions, often under the assumption that those are shared by all involved researchers and sometimes even the relevant academic discipline.

After this first ‘coding sweep’, researchers often want to recode. When the initial coding consisted of indexing (i.e. organising the data based on the topic that was discussed), the coding necessary to address the research question won’t even have started yet — but even when it has, once it’s possible to view all source fragments that were coded with a specific code, patterns often become apparent that were harder to spot before having seen the data grouped by code. In addition, the coding structure can often be improved to represent patterns in the data better.

The choices made during recoding

### 13.2 Deleting codes





## Chapter 14

# The iROCK interface

### 14.1 Background

iROCK is an interface for coding sources that consists of an application built in HTML, CSS and Javascript. This means that it does not store any information on a server, which has as a benefit that its use is necessarily GDPR compliant. This means that even if you have not anonymized your sources, you can still comfortably use iROCK, resting assured that you cannot violate the GDPR by doing so.

iROCK is available at <https://sci-ops.gitlab.io/irock/> - this is a version hosted by GitLab. You can also download all files to your local PC, and run it from there. To do that, simply visit the repository at <https://gitlab.com/sci-ops/irock> and click the download button at the right-hand side (next to the “History” and “Find file” buttons). You can download the archive, unpack it on your PC, store the files somewhere, and then double-click the “index.html” file in the “iROCK” directory.

Note that iROCK is Free/Libre Open Source Software (FLOSS), which means that you can inspect the source code, and add functionality, if desired (see the repository at <https://gitlab.com/sci-ops/irock>).

### 14.2 Using iROCK



## Part III

# Applying the ROCK



# Chapter 15

## A ROCK workflow

This chapter describes, or perhaps more accurately, prescribes, a recommended workflow to use when working with the Reproducible Open Coding Kit (ROCK). Consistent with the aims of the ROCK, this workflow is designed to optimize transparency and reproducibility.

This ROCK workflow leans heavily on the `rock` R package and the iROCK interface, but of course, any of the actions described can be implemented in other ways as well.

### 15.1 A basic ROCK workflow

In qualitative studies where the collected data are already clean and only one coder is used, the workflow is very simple. This workflow is explained first: both because it is all some readers will need, and because it will give an impression of the core elements to those readers who will need the more advanced functionality.

#### 15.1.1 A bit of project management

All projects require some minimal management. When working with computers, this management concerns, among other things, how to organise the related files. When working with qualitative data, there will usually be two types of files: files with participants' personal data, and files without personal data. The latter can safely (and relatively indiscriminately) be synchronized with other computers, while the former requires more care. It is important to have clear procedures for anonymizing data and for making sure the right files are backed up in the right way. For the anonymized transcripts, analysis scripts, and other scientific materials, we recommend using a version control system such as Git. However, file and project management go beyond the scope of this book.

One trick that does fall within the scope of this book is the functionality of R Projects. An R Project is a collection of related files that sit in a directory. In RStudio, you can create an R Project through the New Project menu option in the File menu. You can either create a project in an existing directory; create a fresh project in a new directory; or connect to a version control system such as Git to clone an existing project to your computer (see this chapter in the Psy Ops Guide for more information about Git).

Once you created a project, RStudio will remember which files you had opened in your script file panel (top-left). It also conveniently shows the files and directories in your project in the Files tab of the bottom-right panel, and if you use Git, allows you to synchronize your changes using the Git tab in the top-right panel. Most importantly for our present purposes, using an R Project allows easy access to your files and directories regardless of where on the PC they are located. Therefore, start by creating

an R Project. Once you created it, to continue working on this project, simply open the associated file (with the “.Rproj” extension).

### 15.1.2 Source collection and preparation

We will assume that the data are organised into one or more sources, which are plain-text files where each smallest codable element is placed on a separate line (i.e. each utterance is separated by newline characters). Note that the smallest codable element is not the smallest element that could be coded in theory, but instead represents the smallest element that the researchers are interested in coding. If the data are not yet organised like this, you will first need to clean and organise them; please refer to those sections in the extensive workflow.

In each source, add case identifiers. Case identifiers indicate which utterances belong to which case. Cases are usually data providers, such as participants or organisation. Case identifiers can be, for example, numbers, letters, codes, or pseudonyms. Case identifiers are added to the ROCK like this:

```
[[cid=1]]
```

The “1” is the identifier itself; this could also be, for example, “F23b”, “Alice”, or “F”, depending on the system used to identify the sources. This case identifier is used as an efficient way to attach attributes to the relevant utterances. By default, case identifiers are so-called “persistent identifiers”, which means that once they have been specified on a line, all subsequent utterances in that source will be considered to have been coded with that case identifier, until a new case identifier is encountered.

### 15.1.3 Attribute specification

Cases function as a method for attaching attributes to utterances. For example, if sources are transcripts from interviews, cases can be the participants that were interviewed. This enables attaching participants’ attributes to utterances, such as their age, gender, or area of residence (of course, constructs measured by measurement instruments such as questionnaires can also be used, such as scores on extraversion or self-efficacy).

These attributes can be defined in so-called YAML fragments that are delimited with three dashes (“---”), such as this fragment:

```
---
ROCK_attributes:
-
  caseId: 1
  sex: female
  age: 50s
---
```

Such fragments can be placed in sources (usually at the beginning or the end, although the **rock** doesn’t care), but it may make more sense to combine them all in one separate file. To combine the attribute specifications for multiple cases, simply repeat the same information, including the dash:

```
---
ROCK_attributes:
-
```

```

    caseId: 1
    sex: female
    age: 50s
  -
    caseId: 2
    sex: male
    age: 30s
---
```

Note that when working with YAML, indentation is very important. The word “`ROCK_attributes`” must always start at the beginning of the line; the dash that indicates that attributes for a new case start must always be indented *exactly* two spaces; and the `caseId` and attribute names and their values must always be indented exactly four spaces. If this is violated, the `yaml` package will throw a “Parser error”.

#### 15.1.4 Coding

To code, one can use any text editor able to edit plain text files, such as Notepad, TextEdit, Notepad++, Vim, or BBEdit. However, in this workflow, we will work with iROCK, an interface optimized for working with the ROCK. iROCK is a simple, userfriendly and GDPR-compliant interface for coding sources. The iROCK interface is discussed in detail in Chapter 14. To load it, visit <https://sci-ops.gitlab.io/irock/> in your browser (or follow an alternative method as explained in Chapter 14).

In iROCK, import a source by clicking the red rectangle marked “Sources”. Then, if you want to engage in inductive coding, you can simply start coding. At the right-hand side, you can create codes, and once created, a code can be dragged and dropped from the list onto the utterance you want to apply it to. Click an applied code (in the source) to remove it again. To indicate that a code falls under another code, use the ROCK hierarchy marker: “>”, e.g. “parentCode>childCode”.

If you want to use deductive coding, import your codes by clicking the red rectangle marked “Codes”. You can import a plain text file: every line of the file will be imported as one code. If you already coded one or more sources, you can use the `rock` R package to efficiently create this list from the used codes. First, use the `rock::parse_sources()` function to import the sources. This is explained more in detail in the next section, but looks roughly like this:

```
parsedSources <-
  rock::parse_sources(input = here::here("data", "coded"));
```

Then, use the `rock::export_codes_to_txt()` function to export the codes. For example, to export all codes, including their so-called “paths” (their explicitly specified position in the hierarchy using the ROCK hierarchy marker, “>”, use:

```
rock::export_codes_to_txt(input = parsedSources,
                          output = here::here("codes", "exported-codes.txt"));
```

Alternatively, you can specify that you only want the “leaves” of the code tree, in other words, you don’t want to select codes that have child codes, using “`leavesOnly=TRUE`”, and you can specify that you don’t want to include the path, but instead only want the codes themselves, using “`includePath=FALSE`”:

```
rock::export_codes_to_txt(input = parsedSources,
                          output = here::here("codes", "exported-codes.txt")
                          leavesOnly=TRUE,
                          includePath=FALSE);
```

To only select codes with a given parent, specify a value for “onlyChildrenOf”, and to only select codes that match a given regular expression, specify it as “regex”.

The `rock::export_codes_to_txt()` function will write a plain-text file to disk that can then be directly imported into the iROCK interface.

### 15.1.5 Analysing the results

### 15.1.6 (Re)Coding

### 15.1.7 Publishing the project

## 15.2 An extensive ROCK workflow

Below follows a more extensive workflow description. For the sake of completeness, this also includes common tasks in qualitative research that are unrelated to the ROCK. It is, after all, the extensive workflow.

### 15.2.1 Planning

#### 15.2.1.1 Research questions

Like with any study, it is vital to have a clear research question. The research question determines which methods can be used. For example, not all research questions can be studied using qualitative research (and not all research questions can be studied using quantitative research). Typical research questions that require quantitative research are questions about associations or causality. Typical research questions that require qualitative research are questions about experiences, narratives, and contents of constructs. And if strong conclusions are desired, research syntheses are required, rather than a single study.

The research question is important because once it is clear, the required analysis approach can be determined, which allows determining the required coding approach, which allows determining how to collect the data. Because the research question is so fundamentally connected to all other aspects of the study, one approach to clarify your research question is to think about what potential answers may look like.

One important implication of a research question is whether the coding will be deductive or inductive. Deductive coding uses predefined codes, and inductive coding uses codes created during the coding. Deductive (‘closed’) coding allows more transparency and reproducibility and enables procedures to minimize bias. The price the researcher pays for these advantages is less flexibility during coding. Inductive (‘open’) coding allows identifying patterns and categories in the data that could not be anticipated a priori. In that sense, inductive coding plays to the strengths of qualitative research: it imposes no constraints on analysis. The downside is that it cannot use tools to manage subjectivity; such tools inevitably impose structure and as such decrease the purely inductive nature of the coding.

In practice, coding is often a mix. Researchers rarely start collecting data in a field where no relevant theory exists. Therefore, those theories often shape the coding, in which case making that explicit by



prespecifying a deductive coding structure aids transparency. This coding structure can then form the basis for the coding process, while still allowing coders to add more code trees to the coding structure's root (for codes that cannot be captured by the prespecified codes and their definitions) and to add more codes as 'children' of prespecified codes.

### 15.2.1.2 Coding instructions

Unless there are no preconceived ideas about the coding process whatsoever, the coding process will inevitably require matching utterances to some definition. It is therefore important to have the relevant definitions available, in sufficiently clear and explicit formulations, as well as coding instructions. Coding instructions are important for decreasing undesirable, invisible subjectivity and bias and in the coding process. They explicitly capture the characteristics that a piece of data must satisfy to code it with a given code, including explicit guidelines for resolving edge cases (see section 20.3.2).

It is usually desirable to make sure the coding instructions are consistent over studies. Ideally, the exactly same coding instructions are used in all studies in a lab, department, or even institution (also see section 20.2.1).

If the qualitative study concerns humans, and therefore, the codes relate to constructs (e.g. psychological, sociological, or anthropological constructs), using a decentralized construct taxonomy (DCT) supports clear definitions that can consistently be applied over multiple studies. These are introduced in Chapter 20. For studies with humans, it is therefore strongly recommended to not proceed until a set of DCT specifications has been produced and the coding instructions have been generated from those DCTs.

If coding another type of content, it is still important to develop clear, unequivocal coding instructions before proceeding. The coding instructions should ideally be good enough to render individual coders more or less interchangeable.

### 15.2.1.3 A note on data management

- encryption
- password management

## 15.2.2 Data collection

The operational aspects of data collection vary with the type of data that are collected. We will cover two scenarios here: recording audio from individual interviews, group interviews or focus groups, and collecting existing data such as social media posts or archive materials.

### 15.2.2.1 Recording audio

- 2 recorders (one backup; redundancy)

#### 15.2.2.1.1 Transcription into sources

- with group interviews, pay attention to distinguishing group members; make sure they introduce themselves

### 15.2.2.2 Collecting existing data

## 15.2.3 Source cleaning

Once a dataset has been collected, it is usually necessary to perform some cleaning. In a ROCK workflow, this cleaning includes rudimentary segmentation into *utterances* (see Chapter 10). In the ROCK specification, utterances are separated by a newline character: in other words, every utterance is on its own line. Utterances are the smallest codable unit, and as such, this is not a trivial step. The logic underlying the convention that utterances are separated by newline characters is that although sentences are themselves often hard to fully understand without context, at least they are often self-containing, whereas parts of a sentence are rarely comprehensible on their own.

To clean sources, we will use the `rock` package function `rock::clean_sources()` (for more details, see Section 12.3.1). We assume here that the data are located in a directory called `data` in your Project directory (see Section 15.1.1). In this directory, we assume that the raw sources (i.e. the raw transcripts) are located in the subdirectory called `raw`. In addition, we will write the cleaned sources to another subdirectory of the `data` directory called `cleaned`.

The following command reads all files in the `data/raw` directory in your Project directory, applies the default cleaning operations (e.g. add a newline character following every sentence ending), and writes the cleaned sources to a directory called `data/cleaned` in your Project directory:

```
rock::clean_sources(input = here::here("data", "raw"),
                   output = here::here("data", "cleaned"));
```

Note that if you only want to read files that have a certain extension, such as `.txt` or `.rock`, you can specify this by specifying a regular expression to match against filenames as argument `filenameRegex`. For example, to only read both `.txt` files and `.rock` files, you would pass the regular expression `"\\.txt$|\\.rock$"`, to only read `.txt` files, the regular expression `"\\.txt$"`, and to only read files with the `.rock` extension, you would use this command:

```
rock::clean_sources(input = here::here("data", "raw"),
                   output = here::here("data", "cleaned"),
                   filenameRegex = "\\.(txt|rock)$");
```

The `rock::clean_sources()` function has many other functions, which you can read about by requesting the manual page. You can do this by typing `?rock::clean_sources` in the R console (the bottom-left panel in RStudio).

## 15.2.4 Prepending utterance identifiers

## 15.2.5 Coding and segmentation

### 15.2.5.1 Automating coding and segmentation

In a sense, cleaning codes already applies some automatic segmentation: after all, the default

```
rock::code_sources()
```

### 15.2.6 Manual coding and segmentation

To manually code and segment, any software that can open and save plain text files can be used. To achieve this software independence was, after all, one of the reasons the ROCK was developed. Most operating systems come with basic plain text editors (e.g. notepad on Windows; TextEdit on MacOS; and vim with most Unix systems), and many Free/Libre and Open Source Software (FLOSS) alternatives exist, such as the powerful Notepad++ for Windows and BBEdit for MacOS.

These editors can be used to open sources and add codes and section breaks. Many allow the creation of plugins that can further facilitate this, and in addition, plain text editors can be used in tandem with spreadsheet applications such as the FLOSS LibreOffice Calc. This allows having neatly organised coding structure in a spreadsheet, which can then easily be copied to the clipboard and pasted in a source in a text editor. By using key combinations such as Alt-Tab (Windows) or Command-Tab (MacOS) the coder can quickly switch between the source and the code overview.

In addition, iROCK can be used, a rudimentary graphical user interface that simply allows appending predefined codes to utterances and inserting section breaks (for segmentation). More details are available in Chapter @(*irock*).

Finally, the ROCK standard enables development of a variety of other tools for specific use cases.

### 15.2.7 Inspecting coder consistency

### 15.2.8 Merging sources

```
rock::merge_sources
```

### 15.2.9 Viewing source fragments by code

```
rock::collect_coded_fragments
```

### 15.2.10 Recoding

Sometimes, it is desirable to change codes. For example, a set of codes that was initially obtained through inductive coding may, upon inspection, have a hierarchical structure. When using the ROCK, ideally, the originally coded sources remain in their original state so as to enable scrutiny of the coding process. Instead, the recoding is applied using a command that opens the sources, makes the changes, and then writes them to disk again.

At present, the *rock* R package has four functions to code and recode.

```
rock::search_and_replace_in_sources
```

### 15.2.11 Generating HTML versions

```
rock::export_to_html
```

### 15.2.12 Exporting the coding spreadsheet



## Chapter 16

# A Beginner's Guide to the Reproducible Open Coding Kit

### 16.1 Welcome! Step-by-step, no prior knowledge of R needed!

Hello! If you are here, it probably means that you are interested in conducting transparent and rigorous qualitative research and/or you would like to prepare your data for Epistemic Network Analysis. The Reproducible Open Coding Kit (ROCK) can help you with that! If you have looked into what the ROCK is, you might be intimidated by the terms/expressions it uses or the software and processes it requires. Have no fear, the ROCK can be easily used by people with no prior knowledge of R, and we will prove it to you!

The following is a step-by-step account of how to work with the ROCK for a qualitative or a quantitative ethnographic research project using the rock R package. We disclose these steps assuming that you would like to make your process as transparent as possible, facilitating making your project public once (and if) you are ready to do so. Facilitating Open Science (OS) is something we care about deeply: making our processes public (including, but not limited to operationalization, data, full results, etc.) benefits science in general (as we can learn much from each other's best practices and challenges) and increases reliability, reproducibility, and interoperability (read more about OS [here](#)). OS is not a dichotomy (open or closed), but a spectrum; so, if your project requires the data to remain private or your analysis cannot be fully disclosed, the ROCK package can still help you perform various tasks within your project and disclose as much as ethically possible. (Read more about ROCK functionality [here](#)).

Using the ROCK necessitates some preliminary steps, which will be covered in the first section, *Setting up the Basics*, requiring you to download some software and create a repository. The section on *Data Preparation* will help you specify your directory, add your sources and attributes, and designate your persistent identifiers. The *Coding and Segmentation* section will guide you through the process of coding and segmentation with iROCK and with automated coding. The last section is *Aggregation and Analyses*, which helps you aggregate coded data and perform various analyses.

For the purposes of this tutorial, we assume that you have already decided on a research question, have agreed on a research design, and have specified the operationalization of your project. We urge you to fill out a preregistration form for your project before you begin this tutorial (QE-specific preregistration form draft coming soon!). Even if you do not submit it, this form will help you consider vital aspects of your initiative and help you discuss those with team members. In the tutorial, we presume you have the following aspects of your project operationalized/collected/developed and ready to employ:

- \* Sources (raw data in one or more files, anonymized and ready to be coded)
- \* Metadata/Attributes (data about your data; that is, collected variables on your data providers or sources)
- \* Codes (concepts)

or expressions with which you will be coding your data; these should already be developed provided you are using deductive coding to code your data) \* Segmentation (if you are segmenting your sources, we assume you have ideas on how you will be designating meaningful segments in your data)

If you are not familiar with basic ROCK terminology, please refer to this chapter of the ROCK Book before you begin this step-by-step process. Please note that visual aids in this tutorial were generated from different projects; these will be updated with a more coherent set of visuals later. We would like to thank Judit Nyirő in helping us create this tutorial and checking our instructions for clarity at every step.

Okay, if you are ready, let's begin setting things up!

## 16.2 Setting up the Basics

### 16.2.1 Install R stuff

Let's begin setting up the basics by downloading and installing R and RStudio! You will need both of these to use the ROCK and perform various operations.

#### 16.2.1.1 Download R

For Windows / For Mac

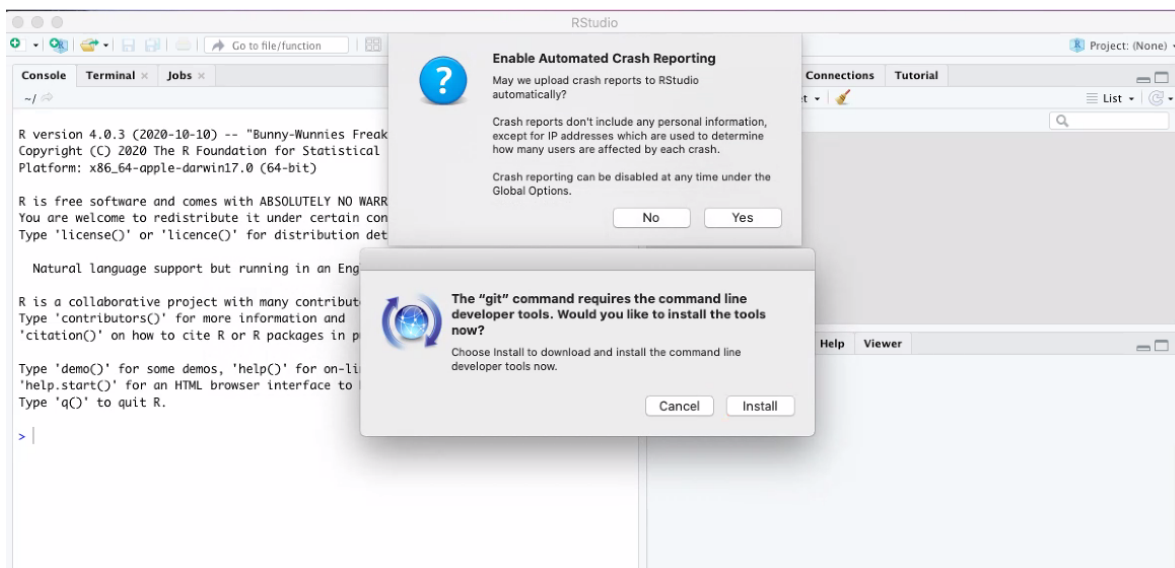
#### 16.2.1.2 Download RStudio

For both windows and Mac

**What is R?** R is a programming language and software environment for statistical computing and graphics. Many programmers around the world use R to create software, which are then made freely available for common use mainly via a platform called CRAN, the Comprehensive R Archive Network. CRAN is a network of servers around the world that store identical, up-to-date versions of R code and documentation. If you want to think of all this metaphorically, you can imagine R as a natural, spoken language like English. People around the world use words in English (elements in the R code) to create shorter or longer strings of sentences (pieces of software). These sentences can be used by researchers like you to e.g. perform analyses on their data, and they can be used by other programmers to modify, expand, build on these sentences to create new pieces of “writing” in R (more complex programs).

**What is RStudio?** RStudio wraps around R and makes life easier. It provides one interface where you can interact with R, work with analysis scripts, view plots and other results, interact with a version control system called Git that we'll discuss later, and RStudio introduces the concept of projects, which are a useful way to bind together a set of files.

When you start using RStudio on MacOS, you may see these windows pop up. Feel free to click on “install” in the bottom window to get all the tools you need. The top window is a personal preference you should consider and then hit “No” or “Yes” accordingly.



## 16.2.2 Create a Gitlab account and project

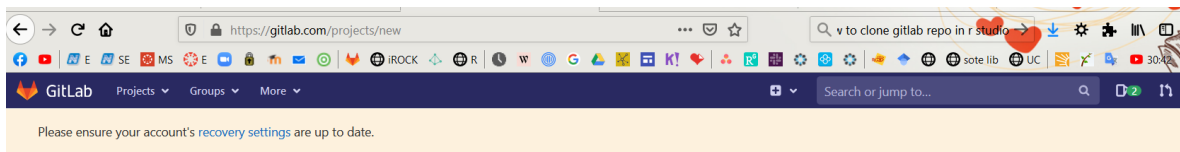
We assume you will be using some kind of repository to store and share your materials. We will demonstrate the process through Gitlab here.

Git is an extremely powerful and relatively user-friendly version control system. You can think of it as the “tracked changes” functionality in word processor software such as Microsoft Word or LibreOffice Writer, but on steroids, and for all files in your project. It was originally developed to facilitate collaborating with many people on a set of files, simultaneously, without running the risk of accidentally losing anything. Like R, Git itself is quite bare, and so many people interact with it through tools that augment it. One such tool is RStudio. Another class of tools are online Git repository managers.

Two popular examples of such online repository managers are GitHub and GitLab. The latter is Open Source, while the former is not; GitLab, therefore, is consistent with the Open Science foundations upon which the ROCK was built, and is what we will use in this tutorial.

The Git integration in RStudio means that you will be able to make modifications to the files in your directory through RStudio (e.g. perform analyses), which are then easily “pushed” to the central GitLab repository from RStudio. You are in essence performing actions on your PC, which can then be “synced” with your (public) repository on Gitlab.

Firstly, please create a GitLab account, if you don’t already have one. You can sign up here: [https://gitlab.com/users/sign\\_up](https://gitlab.com/users/sign_up) Then, go to “Projects” and “Your projects” and then click on “New project”



Fill in the details as instructed by GitLab and according to your preferences, then click “Create project”. Select the option “Initialize repository with a README”. Where you see “visibility level”, please note that the default is “Private” and switch this to “Public” (unless you don’t want to publish the repository, or want to publish it later on).

New project > Create blank project

**Create blank project**  
Create a blank project to house your files, plan your work, and collaborate on code, among other things.

**Project name**  
My awesome project

**Project URL**  
https://gitlab.com/

**Project slug**  
my-awesome-project

Want to house several dependent projects under the same namespace? [Create a group.](#)

**Project description (optional)**  
Description format

**Visibility Level** ⓘ

☒ Private  
Project access must be granted explicitly to each user. If this project is part of a group, access will be granted to members of the group.

☐ Public  
The project can be accessed without any authentication.

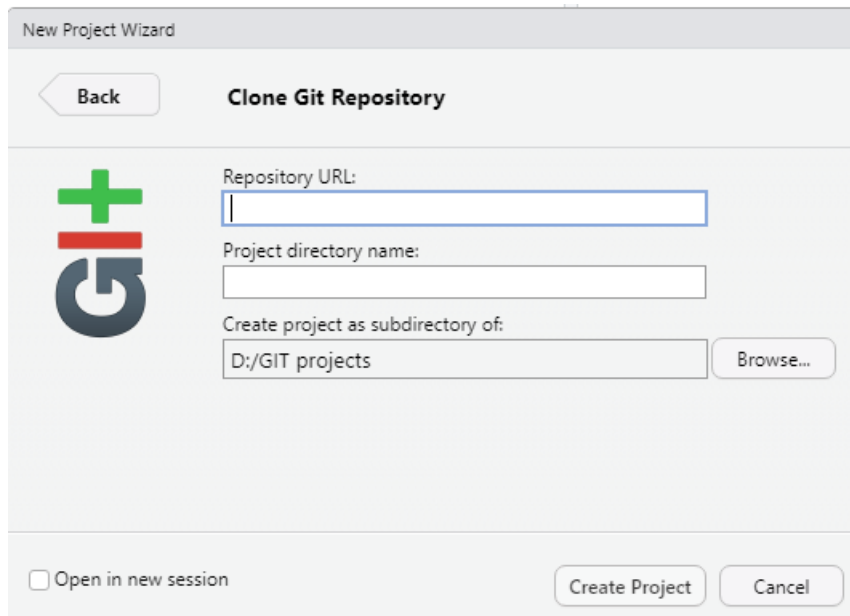
☐ **Initialize repository with a README**  
Allows you to immediately clone this project's repository. Skip this if you plan to push up an existing repository.

[Create project](#) [Cancel](#)

### 16.2.3 Create an R project

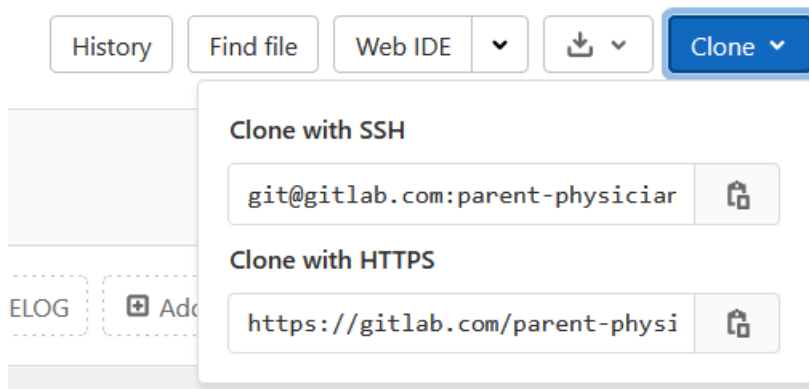
Open RStudio, click on “File” and then “New project”. Click on “Version control” and then “Git”.



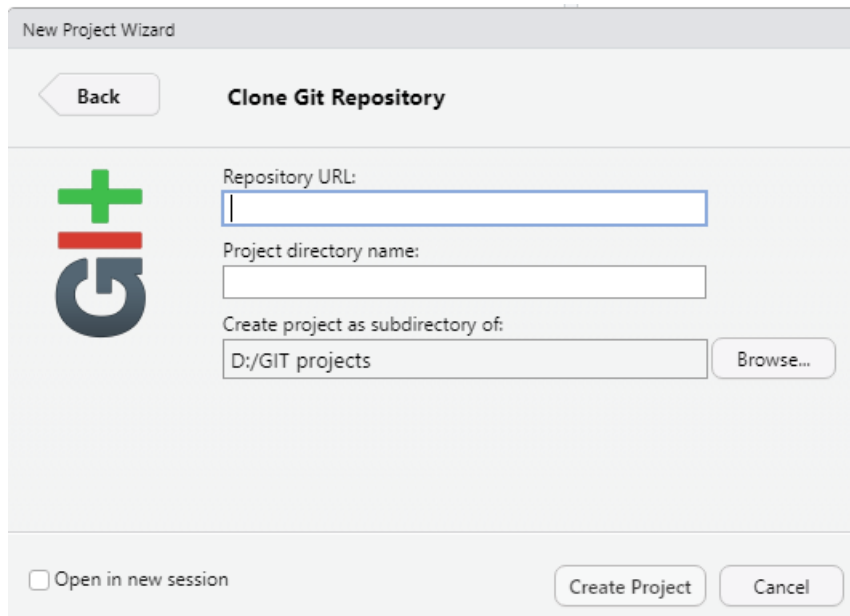


Now you will see a field at the top in which you can enter the Git HTTPS cloning URL of your Git project. To find it, you will need to go to your Gitlab project's page.

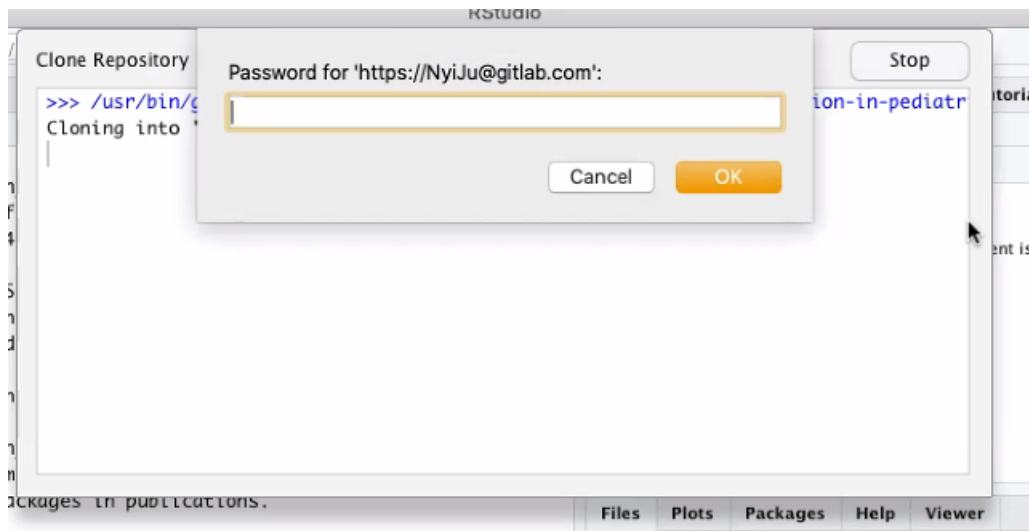
Go to Gitlab, find your project, and click on “Clone”. Click on the “copy” icon to the right of “Clone with HTTPS” or just highlight the URL and click Ctrl+C to copy it.



Take this URL back to RStudio and enter it into the field “Repository URL” (you can just hit Ctrl+V). The field below it, called “Project directory name”, will be autofilled, either upon pasting the URL, or as you hit Tab or click the next field. The last thing you need to do is designate into which MAIN directory this particular project should be a SUBDIRECTORY of. Note that Git will create a directory with the “Project directory name”. We suggest you have a main directory for all your projects (e.g. “research” or “studies”) and in that case, we would now select that directory and RStudio will create a subdirectory within that. When finished, click “Create project”.



Cloning is always possible for public projects, but for private projects, you will need to authenticate to the GitLab server. RStudio will usually present a dialog requesting the necessary information. It will look something like this:



It will ask you for your Gitlab username and password.

If you are not prompted, you may receive an error message of some kind, asking you to specify these. In this case, enter your information below and copy-paste the text into your console and hit enter:

```
git config --global user.name "YourUserNameHere"
git config --global user.email "YourEmail@Here"
```

This should prompt RStudio to ask for your password if you retry. If it does not, you may have specified an incorrect password and will need to reset this. The way this works depends on which credential manager you use – see <https://psy-ops.com/git-basics#git-unsetting-password> for some pointers.

Congratulations, you now have an R Project, which is mostly empty right now, but we will start populating it soon!


### 16.2.4 Copy the Empty ROCK project and personalize it

The easiest way to prepare for using the ROCK is by copying the “Empty ROCK project” repository.

Go here: <https://gitlab.com/psy-ops/empty-rock-project>


Download as a zip file to anywhere on your PC, and then unzip it.



psy-ops > Empty ROCK project > Details


**Empty ROCK project**
Project ID: 16681566 | [Request Access](#)

34 Commits
1 Branch
0 Tags
287 KB Files
12.1 MB Storage

This is an empty project for a study using the Reproducible Open Coding Kit (the ROCK).

master
empty-rock-project / +
History
Find file
Web IDE

Close

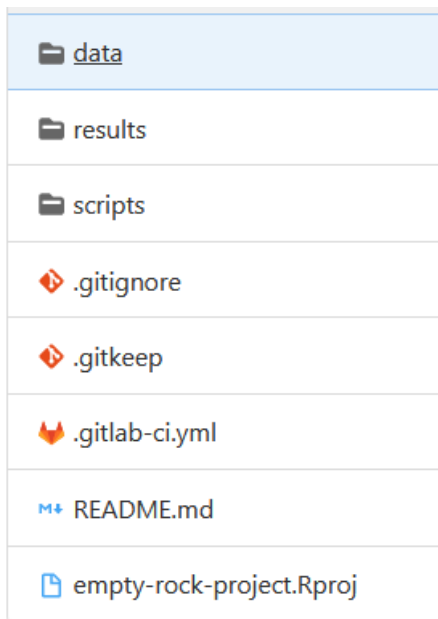

**Slightly more content**  
 Gjalt-Jorn Peters authored 6 months ago
 
ff04a47d

README
CI/CD configuration
No license. All rights reserved

Name	Last commit	Last update
data	Added anonymization	6 months ago
results	Added some stuff	7 months ago
scripts	Slightly more content	6 months ago

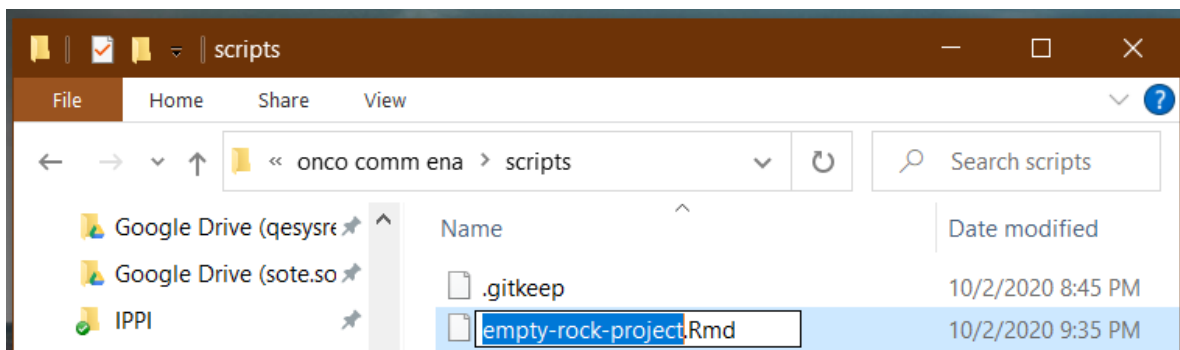
Place the contents of the unzipped folder into the directory you chose for the R project that you created when you cloned your GitLab repository. Please note that if you ever move this directory, you will need to open it in RStudio from the new location before you can open it from within RStudio. When copying these files, your computer will probably warn you that one or more files already exist. You can safely skip these. For example, you already created an empty “README.md” file for your project, so you won’t want to overwrite that with the Empty ROCK project README file.

Below is the main directory of the Empty ROCK project; feel free to delete the “empty-rock-project.RProj” file (the RStudio project; as you have your own RStudio project, completed in the previous step).

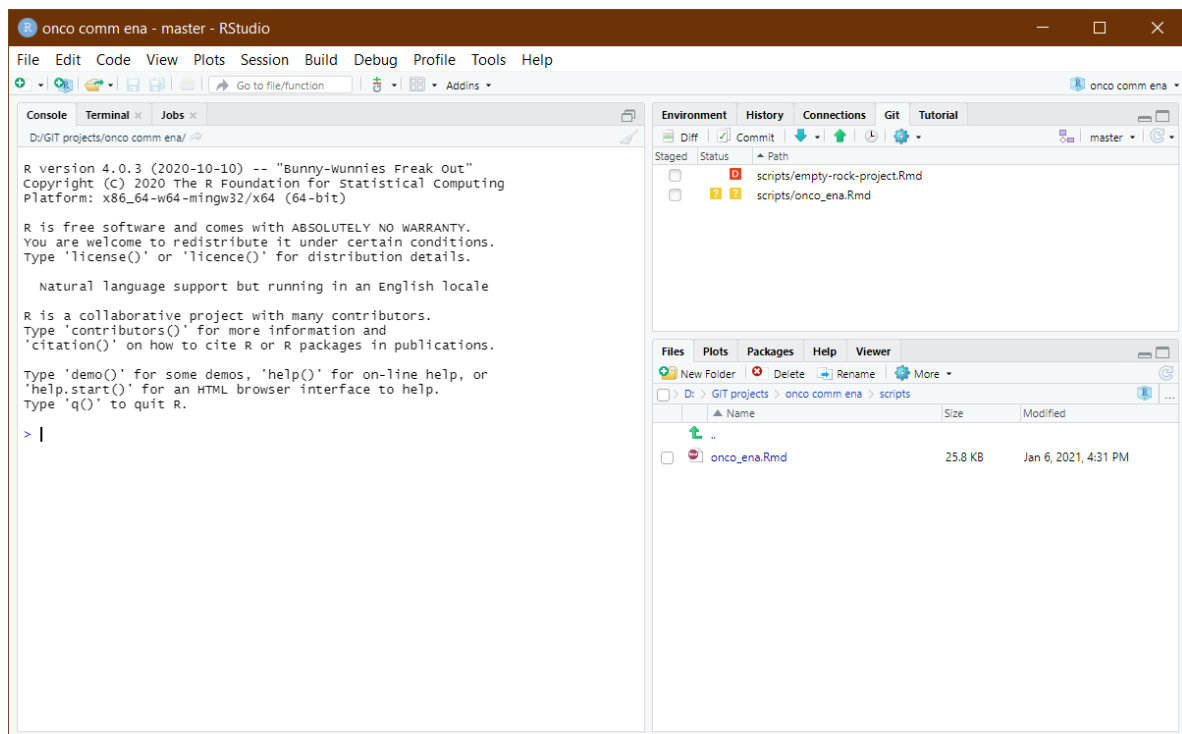


The “data” directory will be where your raw and coded data go; the “results” directory is for your intermediate and final results. The directory called “scripts” contains the R script(s) you will be using and modifying throughout your project. The “README.md” file contains the content that appears on the main page of the project in GitLab (if you scroll down below the overview with all the files in your repo). The three files in red all beginning with “git” will be discussed a bit later (xx).

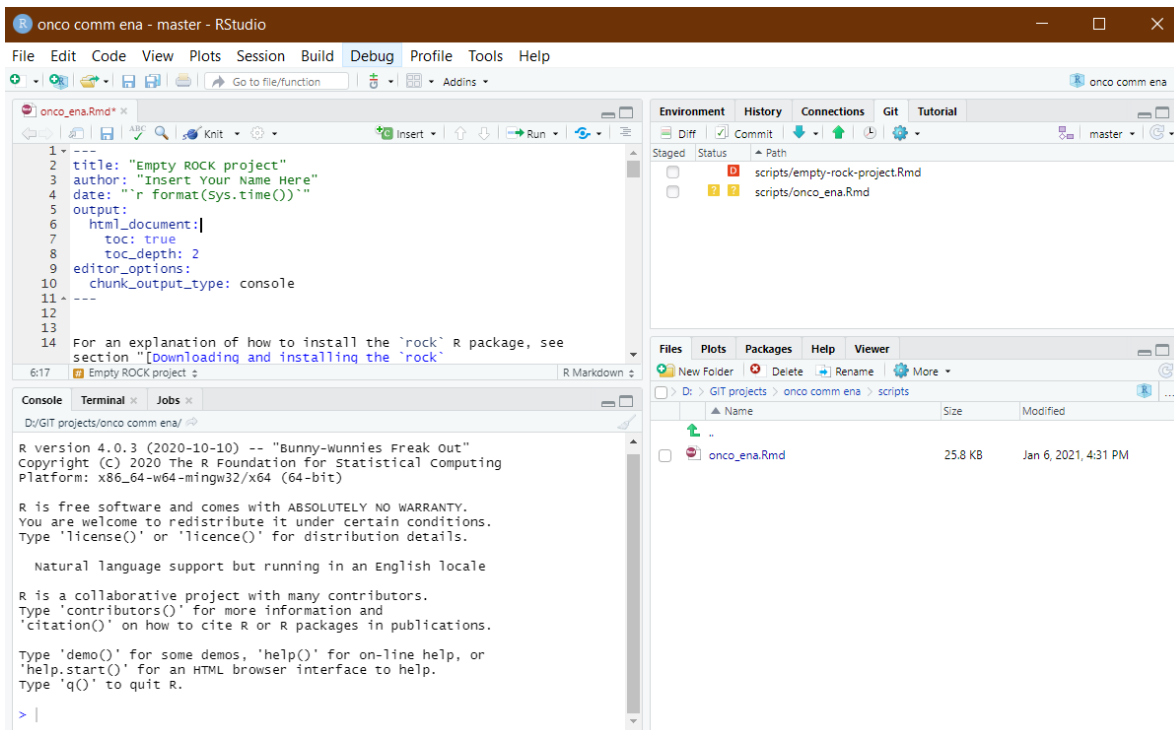
In the “scripts” directory, rename the file according to your preferences. We suggest naming it the same as or resembling the entire RStudio project.



Now, in the main directory, double click on the RStudio project: this will open it in RStudio. Take a look at the RStudio interface.



The pane on the left is called the “console”, this enables you to communicate with the R software environment. All panes are multifunctional: for example, in the bottom-right pane you can see there are tabs, usually with the “Files” tab opened. This shows you where you currently are (“D:/Git projects/onco comm ena/scripts”; this is where we have placed our Git repo, and in it, the Empty ROCK project contents). In the bottom right pane, if we click on the “scripts” directory and then on the so-called R Markdown file (with extension .Rmd; note that depending on your computer’s settings, file extensions may be hidden from you), which contains our R script, it will open in a pane above the console.



The script (Rmd file) contains some explanations and commands (instructions for R to do things with your data) that you will most likely need for your project. If you scroll down or drag the divider between the script and the console down, you will see more of the script file. An R Markdown file combines chunks or R code (normally stored in R script files with the extension “.R”) with text in the markdown format (normally stored in plain text files with the extension “.md”). In this file, the text that is already present is formatted using that markdown format and supplies you with background and instructions on the ROCK, and is interspersed with R chunks that useful commands. The markdown format uses hashtags (“#”) for headings, one hashtag indicates a level 1 heading, two hashtags a level 2 heading and so on. R commands look different; they appear in a grey box, as shown below (between lines 35-41). Also notice that there is a small green “play” sign in the top right corner of the grey box; clicking on this executes all the commands in that particular grey chunk. We will come back to more features of the script and RStudio later.

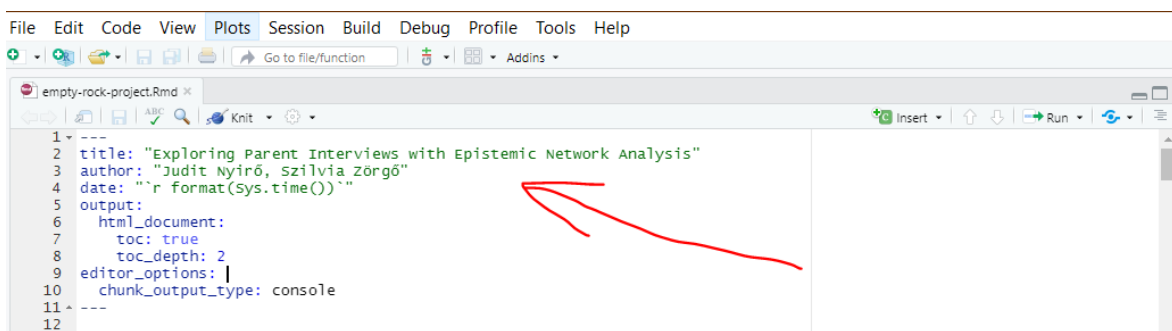
```

25 ## Organizing your project {#organizing-your-project}
26
27 There are basically two approaches to organising the source files. The first is to create new directories ('folders')
28 for each action that changes the sources. In this approach, one directory contains all raw sources; a second directory
29 contains all sources after the 'rock' command to clean them was executed; a third directory contains all sources after
30 the 'rock' command to add the Utterance Identifiers was executed; et cetera. The advantage of this approach is that
31 every directory is in itself a snapshot of the data at the corresponding stage in the project. A disadvantage is that
32 you may end with quite a lot of directories.
33
34 The second approach is to create the files in the same directory, but to use a convention to rename them. In this
35 approach, some string of characters is added to the sources' filenames for each stage in the project. In this
36 approach, the raw sources all have their original name; when the 'rock' command to clean them is executed, it stores
37 its result in filenames that have, for example, "_clean" appended; when the 'rock' command to add the Utterance
38 Identifiers is executed, it appends, for example, "_uids", et cetera. The drawbacks of this approach are that you end
39 up with a lot of files, and very long filenames.
40
41 In practice, what works best will depends on individual preference. We recommend combining both approaches: appending
42 something to the filenames and storing the files of each stage in a different directory. This approach makes it
43 easiest for others to follow what you did. Therefore, that is the approach implemented in this bare bones project.
44
45 For example, look at the directories in this project:
46
47 ```{r echo=FALSE}
48 cat(tail(grep("+-+ public",
49             capture.output(fs::dir_tree(here::here(),
50                                       type="directory")),
51       fixed = TRUE, invert = TRUE, value = TRUE),
52     -1), sep = "\n");
53 ```
54
55 The directories within the 'data' directory all start with double digits. These ensure the directories are shown in
56 the right order (note that these directory names apply the conventions laid out in the \[Psy Ops Guide\]
57 (https://psy-ops.com/files-and-directories); we recommend you do the same). The reason double digits are used is
58 that this allows structuring of the directories in phases. Directories holding sources from preparatory steps start
59 with a 0; directories holding sources from automatic coding steps start with 1; and directories holding sources from
60 manual coding and recoding start with 2.
61
62 ## Organizing the files

```

### 16.2.5 Change the basic info in the script

As a final step for now, rewrite the basic information at the top to reflect the details of your project. The title and author can be changed at any time, do not worry about making these final. This front-matter is formatted in a language called YAML, which has a special format, aspects of which we will be addressing later on. For now, it is important to remember that spaces, hashtags, line breaks, etc. are there for a reason. Try to only re-write or delete characters that you are sure you will not need in the future, and avoid changing the indentation. RStudio uses colors to designate field names and contents, and so should hint at accidentally introduced errors.



This is basically what we will be doing for the rest of the project as well. This basic script contains all the instructions and commands you will need to use the ROCK. You will have to change some things, though. These changes will have to be made directly in this script file by writing over some things within commands. don't worry, we will go into detail on this for each command. For now, just familiarize yourself with how the interface looks and what the different panes contain.

Click "Save"; it is at the top left corner of the pane.

Just to summarize, this R Markdown file contains text in three different languages:

- The front-matter uses "YAML";

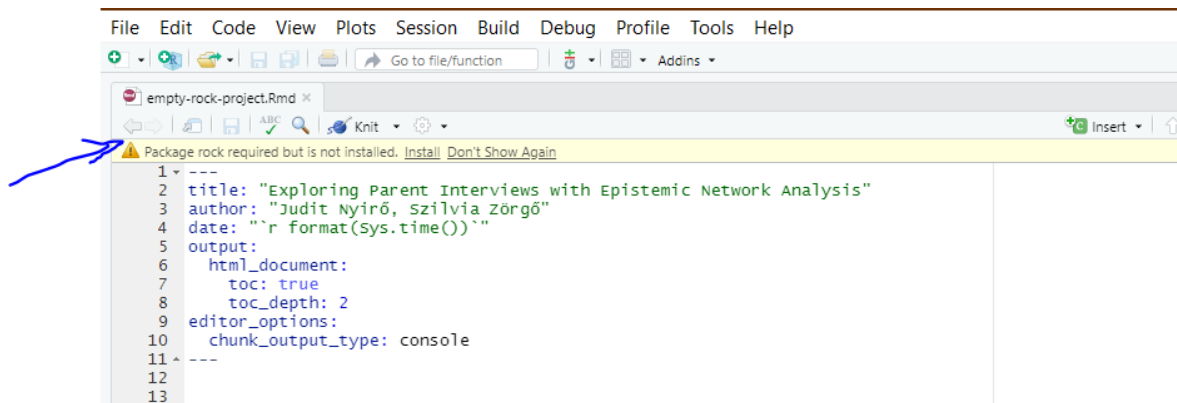
- The text uses Markdown;
- The code chunks use R

It is important to realize that both these languages themselves and the “costs” of deviating from those languages’ conventions differ for each type: for example, errors in the YAML will not interfere with running the R code chunks, but will prevent you from “rendering” the complete file (we will get back to this later); errors in the markdown will in most cases simply ruin the layout of your text, but have no serious consequences; and errors in R code will prevent RStudio from executing your analyses.

In the remainder of this tutorial, we will explain the bare minimum of what you will need to know of these conventions, but it is easy to find a host of tutorials on each of these three (e.g. see xx) should you wish to learn more.

## 16.2.6 Download the ROCK R Package

Open the R project and click on the script file in the lower right pane. You might notice that RStudio has a keen eye for R packages that are in a script and yet not installed on your computer. RStudio will most likely tell you this with a thin yellow ribbon that pops up above the opened script. We suggest you click on “Install” and let RStudio install the necessary packages for you.



Generally speaking, certain actions require certain packages, and packages usually build on other packages. RStudio is very smart and will let you know if you need to install a package that you haven't yet, so whenever you see the yellow ribbon, trust it, click install.

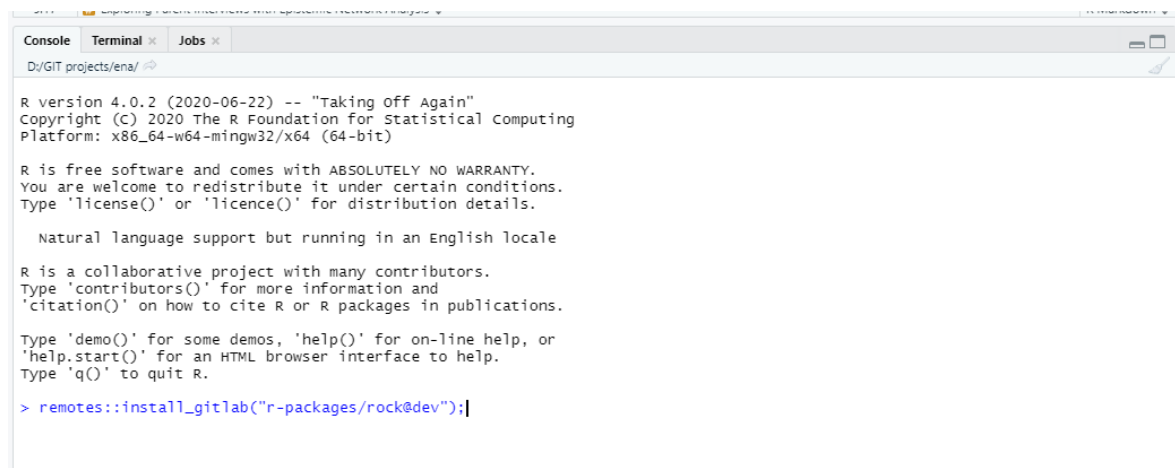
Next, we will install the R package **rock**. Depending on which version you want to install, copy one of the following three commands:

```
install.packages("rock");
remotes::install_gitlab("r-packages/rock");
remotes::install_gitlab("r-packages/rock@dev");
```

(Commands and their context can be found here; note that the latter two require you to have the **remotes** package installed, which you can install using the `install_packages()` function used on the first of these three lines)

Then paste the command into the console in RStudio (bottom left pane) and hit enter.





```

R version 4.0.2 (2020-06-22) -- "Taking off Again"
Copyright (C) 2020 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

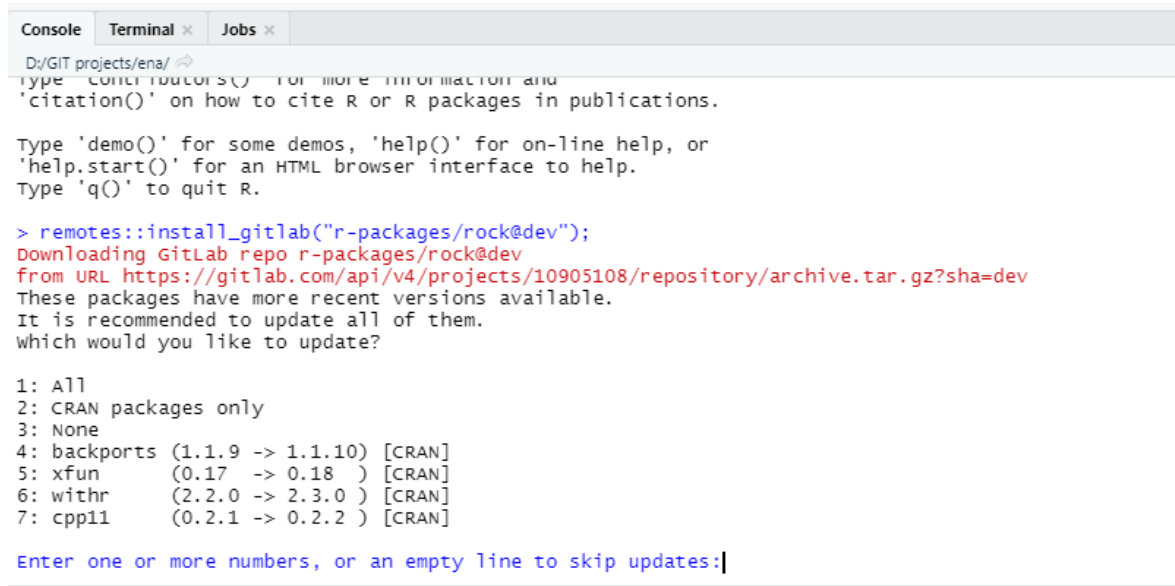
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> remotes::install_gitlab("r-packages/rock@dev");|

```

You might see a list of updates. Choose your preferences to update any or all packages shown below. Then hit enter again.



```

D:\GIT projects\ena\
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> remotes::install_gitlab("r-packages/rock@dev");
Downloading GitLab repo r-packages/rock@dev
from URL https://gitlab.com/api/v4/projects/10905108/repository/archive.tar.gz?sha=dev
These packages have more recent versions available.
It is recommended to update all of them.
which would you like to update?

1: All
2: CRAN packages only
3: None
4: backports (1.1.9 -> 1.1.10) [CRAN]
5: xfun (0.17 -> 0.18 ) [CRAN]
6: withr (2.2.0 -> 2.3.0 ) [CRAN]
7: cpp11 (0.2.1 -> 0.2.2 ) [CRAN]

Enter one or more numbers, or an empty line to skip updates:|

```

### 16.2.7 Update the ROCK R Package

Make sure you have the latest versions of the necessary packages. To do that, find “## Basic setup in R” (here, i.e. in the figure below, on lines 58-82) in the script and run the chunk by clicking on the little green play button on the right. Alternatively, you can highlight the commands and hit Ctrl+Enter. This will run the updates.

onco comm ena - master - RStudio

File Edit Code View Plots Session Build Debug Profile Tools Help

Go to file/function Addins

onco\_ena.Rmd

```

53- ```{r pre-setup, ref.label="pre-setup-specification", echo=FALSE}
54- ### This R Markdown chunk includes the 'quietGitLabUpdate' function definition
55- ### here, allowing that code to be placed all the way at the end of this file.
56-
57-
58- ## Basic setup in R
59-
60- This R code does some basic setup tasks. It installs the most recent versions of the 'ufs', 'yum', and 'rock' R
61- packages; checks for presence of the 'here' package (if it isn't installed, install it using
62- 'install.packages('here');') ; and finally sets the 'knitr' chunk option 'echo' to 'TRUE', which means that by
63- default, each chunk's R code will be included in the rendered HTML file.
64-
65- ```{r}
66- if (!('installed.packages()['ufs', 'version'] >= "0.4")) {
67-   stop("You need to have at least version 0.4 of the 'ufs' package installed; ",
68-        "install it with:\n\ninstall.packages('ufs');");
69- }
70-
71- ### Get most recent versions of some packages, and the development
72- ### version of the rock package - set quiet to FALSE to see info
73- quiet = TRUE;
74- ufs::quietGitLabUpdate("r-packages/ufs", quiet=quiet);
75- ufs::quietGitLabUpdate("r-packages/yum", quiet=quiet);
76- ufs::quietGitLabUpdate("r-packages/rock@dev", quiet=quiet);
77-
78- ### Get additional packages
79- ufs::checkPkgs('here');
80-
81- ### Set options
82- knitr::opts_chunk$set(echo = TRUE,
83-                       comment = "");
84-
85-
86- ## Configuring the 'rock' package
87-
88- The ROCK package has a number of defaults that can be customized. This customization happens in this section. A
89- complete overview of the default settings can be obtained by running 'rock::opts$defaults'.
90-
91-
92-
93-
94-
95-
96-
97-
98-
99-
100-
101-
102-
103-
104-
105-
106-
107-
108-
109-
110-
111-
112-
113-
114-
115-
116-
117-
118-
119-
120-
121-
122-
123-
124-
125-
126-
127-
128-
129-
130-
131-
132-
133-
134-
135-
136-
137-
138-
139-
140-
141-
142-
143-
144-
145-
146-
147-
148-
149-
150-
151-
152-
153-
154-
155-
156-
157-
158-
159-
160-
161-
162-
163-
164-
165-
166-
167-
168-
169-
170-
171-
172-
173-
174-
175-
176-
177-
178-
179-
180-
181-
182-
183-
184-
185-
186-
187-
188-
189-
190-
191-
192-
193-
194-
195-
196-
197-
198-
199-
200-
201-
202-
203-
204-
205-
206-
207-
208-
209-
210-
211-
212-
213-
214-
215-
216-
217-
218-
219-
220-
221-
222-
223-
224-
225-
226-
227-
228-
229-
230-
231-
232-
233-
234-
235-
236-
237-
238-
239-
240-
241-
242-
243-
244-
245-
246-
247-
248-
249-
250-
251-
252-
253-
254-
255-
256-
257-
258-
259-
260-
261-
262-
263-
264-
265-
266-
267-
268-
269-
270-
271-
272-
273-
274-
275-
276-
277-
278-
279-
280-
281-
282-
283-
284-
285-
286-
287-
288-
289-
290-
291-
292-
293-
294-
295-
296-
297-
298-
299-
300-
301-
302-
303-
304-
305-
306-
307-
308-
309-
310-
311-
312-
313-
314-
315-
316-
317-
318-
319-
320-
321-
322-
323-
324-
325-
326-
327-
328-
329-
330-
331-
332-
333-
334-
335-
336-
337-
338-
339-
340-
341-
342-
343-
344-
345-
346-
347-
348-
349-
350-
351-
352-
353-
354-
355-
356-
357-
358-
359-
360-
361-
362-
363-
364-
365-
366-
367-
368-
369-
370-
371-
372-
373-
374-
375-
376-
377-
378-
379-
380-
381-
382-
383-
384-
385-
386-
387-
388-
389-
390-
391-
392-
393-
394-
395-
396-
397-
398-
399-
400-
401-
402-
403-
404-
405-
406-
407-
408-
409-
410-
411-
412-
413-
414-
415-
416-
417-
418-
419-
420-
421-
422-
423-
424-
425-
426-
427-
428-
429-
430-
431-
432-
433-
434-
435-
436-
437-
438-
439-
440-
441-
442-
443-
444-
445-
446-
447-
448-
449-
450-
451-
452-
453-
454-
455-
456-
457-
458-
459-
460-
461-
462-
463-
464-
465-
466-
467-
468-
469-
470-
471-
472-
473-
474-
475-
476-
477-
478-
479-
480-
481-
482-
483-
484-
485-
486-
487-
488-
489-
490-
491-
492-
493-
494-
495-
496-
497-
498-
499-
500-
501-
502-
503-
504-
505-
506-
507-
508-
509-
510-
511-
512-
513-
514-
515-
516-
517-
518-
519-
520-
521-
522-
523-
524-
525-
526-
527-
528-
529-
530-
531-
532-
533-
534-
535-
536-
537-
538-
539-
540-
541-
542-
543-
544-
545-
546-
547-
548-
549-
550-
551-
552-
553-
554-
555-
556-
557-
558-
559-
560-
561-
562-
563-
564-
565-
566-
567-
568-
569-
570-
571-
572-
573-
574-
575-
576-
577-
578-
579-
580-
581-
582-
583-
584-
585-
586-
587-
588-
589-
590-
591-
592-
593-
594-
595-
596-
597-
598-
599-
600-
601-
602-
603-
604-
605-
606-
607-
608-
609-
610-
611-
612-
613-
614-
615-
616-
617-
618-
619-
620-
621-
622-
623-
624-
625-
626-
627-
628-
629-
630-
631-
632-
633-
634-
635-
636-
637-
638-
639-
640-
641-
642-
643-
644-
645-
646-
647-
648-
649-
650-
651-
652-
653-
654-
655-
656-
657-
658-
659-
660-
661-
662-
663-
664-
665-
666-
667-
668-
669-
670-
671-
672-
673-
674-
675-
676-
677-
678-
679-
680-
681-
682-
683-
684-
685-
686-
687-
688-
689-
690-
691-
692-
693-
694-
695-
696-
697-
698-
699-
700-
701-
702-
703-
704-
705-
706-
707-
708-
709-
710-
711-
712-
713-
714-
715-
716-
717-
718-
719-
720-
721-
722-
723-
724-
725-
726-
727-
728-
729-
730-
731-
732-
733-
734-
735-
736-
737-
738-
739-
740-
741-
742-
743-
744-
745-
746-
747-
748-
749-
750-
751-
752-
753-
754-
755-
756-
757-
758-
759-
760-
761-
762-
763-
764-
765-
766-
767-
768-
769-
770-
771-
772-
773-
774-
775-
776-
777-
778-
779-
780-
781-
782-
783-
784-
785-
786-
787-
788-
789-
790-
791-
792-
793-
794-
795-
796-
797-
798-
799-
800-
801-
802-
803-
804-
805-
806-
807-
808-
809-
810-
811-
812-
813-
814-
815-
816-
817-
818-
819-
820-
821-
822-
823-
824-
825-
826-
827-
828-
829-
830-
831-
832-
833-
834-
835-
836-
837-
838-
839-
840-
841-
842-
843-
844-
845-
846-
847-
848-
849-
850-
851-
852-
853-
854-
855-
856-
857-
858-
859-
860-
861-
862-
863-
864-
865-
866-
867-
868-
869-
870-
871-
872-
873-
874-
875-
876-
877-
878-
879-
880-
881-
882-
883-
884-
885-
886-
887-
888-
889-
890-
891-
892-
893-
894-
895-
896-
897-
898-
899-
900-
901-
902-
903-
904-
905-
906-
907-
908-
909-
910-
911-
912-
913-
914-
915-
916-
917-
918-
919-
920-
921-
922-
923-
924-
925-
926-
927-
928-
929-
930-
931-
932-
933-
934-
935-
936-
937-
938-
939-
940-
941-
942-
943-
944-
945-
946-
947-
948-
949-
950-
951-
952-
953-
954-
955-
956-
957-
958-
959-
960-
961-
962-
963-
964-
965-
966-
967-
968-
969-
970-
971-
972-
973-
974-
975-
976-
977-
978-
979-
980-
981-
982-
983-
984-
985-
986-
987-
988-
989-
990-
991-
992-
993-
994-
995-
996-
997-
998-
999-
1000-

```

115:32 Chunk 5

R Markdown

Console Terminal Jobs

D:/GIT projects/onco comm ena/

```

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

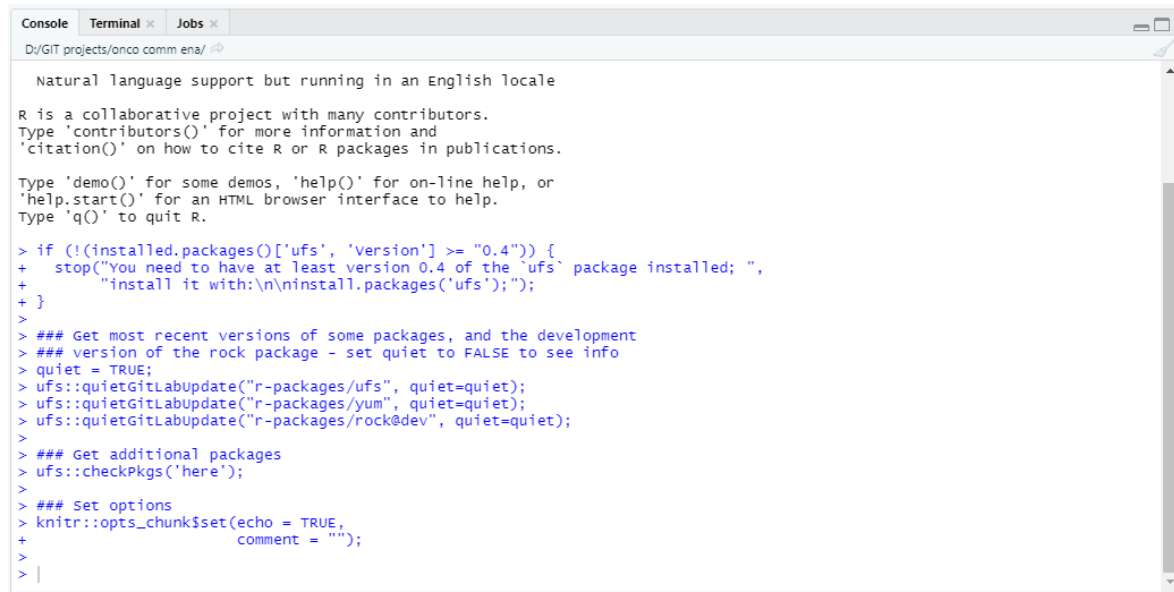
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> |

```

After running the chunk, you should see this in the console:



```

Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

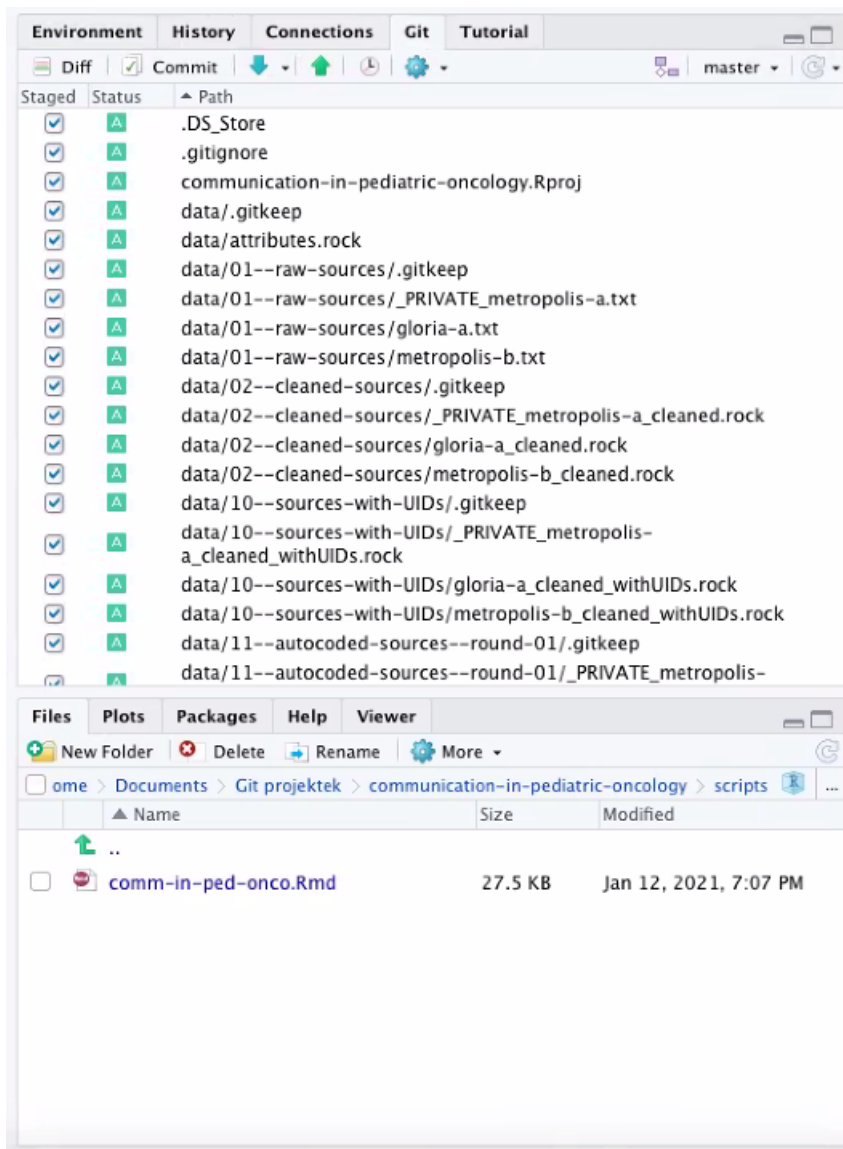
> if (!(installed.packages()['ufs', 'version'] >= "0.4")) {
+   stop("You need to have at least version 0.4 of the 'ufs' package installed; ",
+       "install it with:\n\ninstall.packages('ufs');");
+ }
>
> ### Get most recent versions of some packages, and the development
> ### version of the rock package - set quiet to FALSE to see info
> quiet = TRUE;
> ufs::quietGitLabupdate("r-packages/ufs", quiet=quiet);
> ufs::quietGitLabupdate("r-packages/yum", quiet=quiet);
> ufs::quietGitLabupdate("r-packages/rock@dev", quiet=quiet);
>
> ### Get additional packages
> ufs::checkPkgs('here');
>
> ### Set options
> knitr::opts_chunk$set(echo = TRUE,
+                       comment = "");
>
> |

```

Great, you are now up-to-date! Please note that since the ROCK is being developed continuously, you may need to update it every once in a while, especially when using a new command.

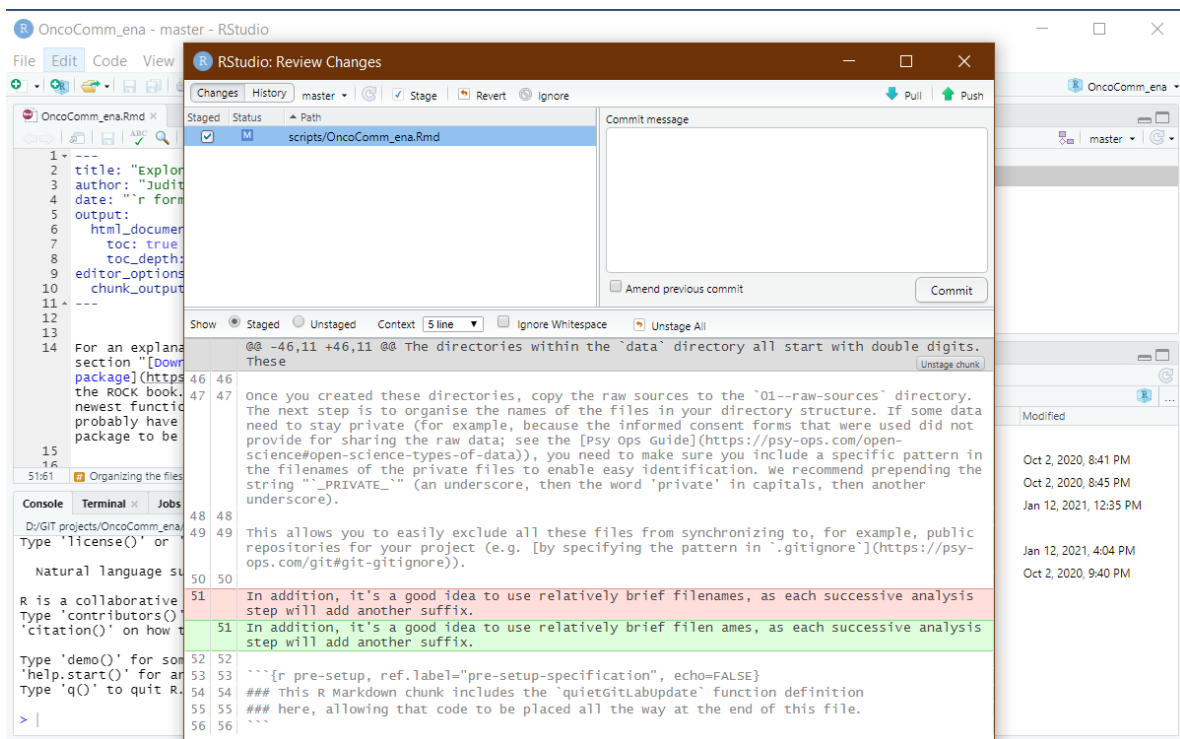
## 16.2.8 First “push” to GitLab

Open your RStudio project, if you do not presently have it opened. We have already made some changes at the top of our script (title, author). Any changes you make on your PC, whether they be made through RStudio or any other program, RStudio will see which files have been modified and what those modifications were exactly; these will appear in the top right pane in the “Git” tab. For example:

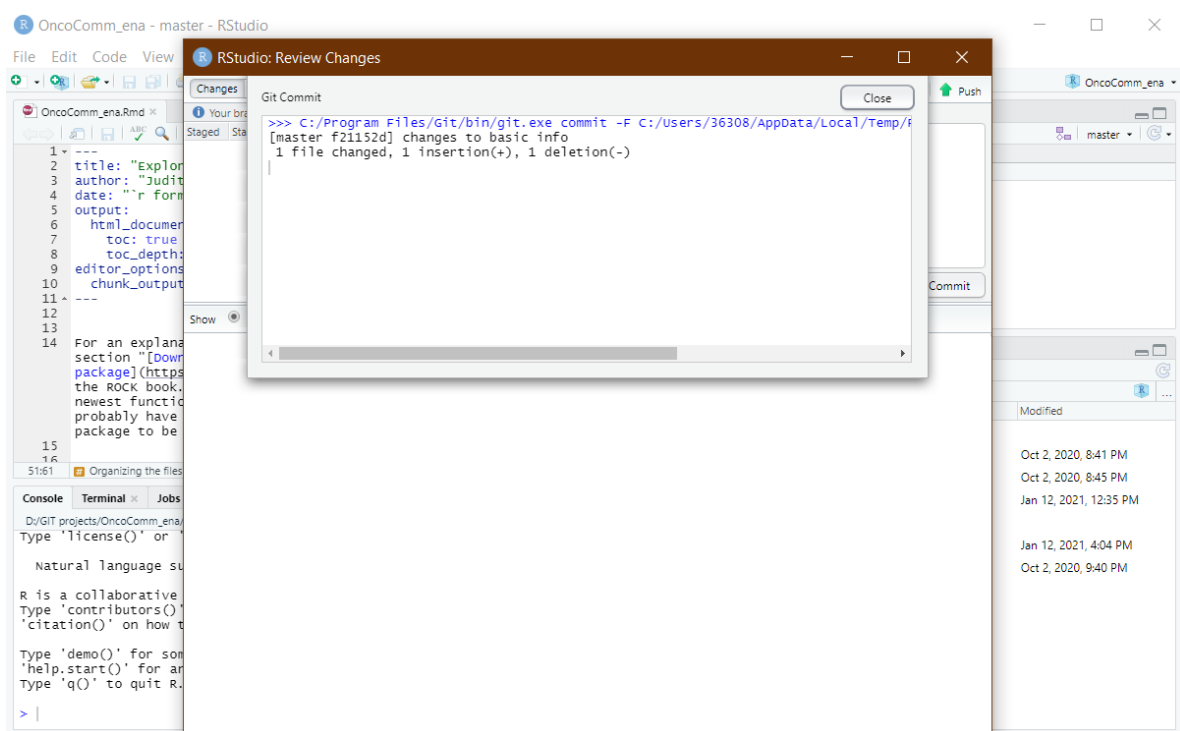


Each time you make changes, the names of the files that were modified will appear in the top right pane (here: “OncoComm\_ena” in folder “scripts”). Remember, RStudio communicates with the GitLab repository. In order to allow these changes to appear in the repository, you first need to “Commit” these changes, then “Push” them to GitLab. To put it another way: there is now a channel between RStudio (your PC) and GitLab (your repository), but the floodgate is closed for your benefit. Once you are sure the modifications you made to your files are ready, you can open the floodgate by “pushing” to Git. This ensures that you have control over each version of your files.

Make sure you are on the “Git” tab of the upper right pane. Click the small box directly to the left of the name of the modified file, then click “commit”. A window will pop-up, letting you review the changes that you’ve made to each file. When ready, enter a short message in the “Commit message” box on the right, then click “Commit”.



Following this, you will see a record of the changes that you have accepted appearing in a new, overlaid pop-up window. Click “Close” and then on the “Review Changes” pop-up window (this should be directly underneath the “Git Commit” pop-up that you just closed), click on “Push”. This will actually push the changes you have accepted to your repository; don’t worry if there is nothing in that window...you still need to click on “Push”. You can now click “Close” for both pop-up windows and return to the main RStudio interface.



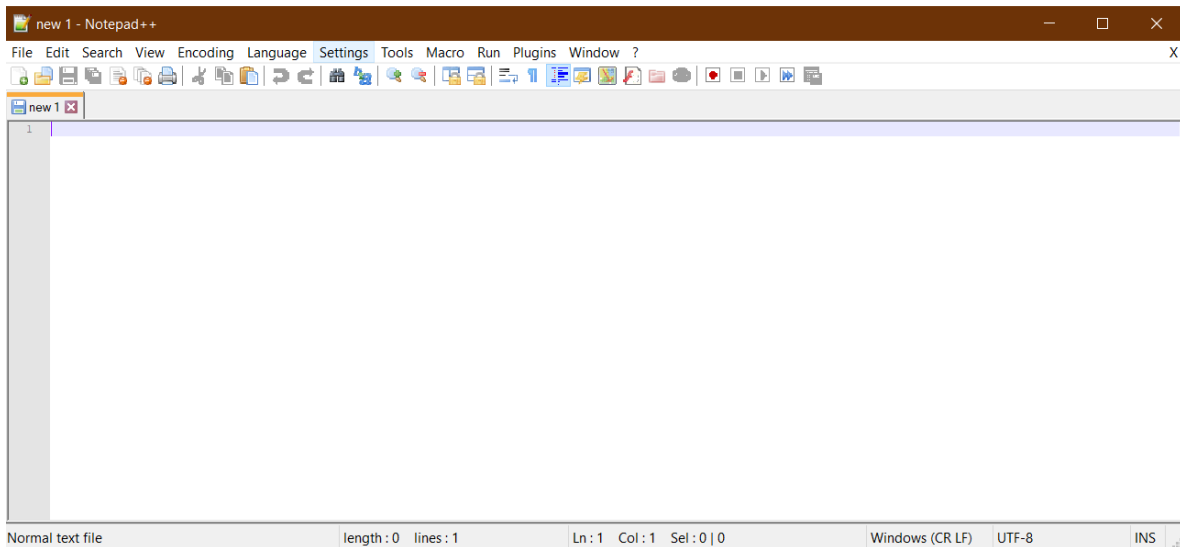
Congratulations! You have now pushed your changes to your repository! You can go to Gitlab, refresh

the page, and make sure your changes are actually there.

## 16.2.9 Download Notepad++

You will need a text editor, so we suggest you download Notepad++, which works very well with multiple file types and encodings. And its logo is super cute. Download and install the most recent version here: <https://notepad-plus-plus.org/downloads/>

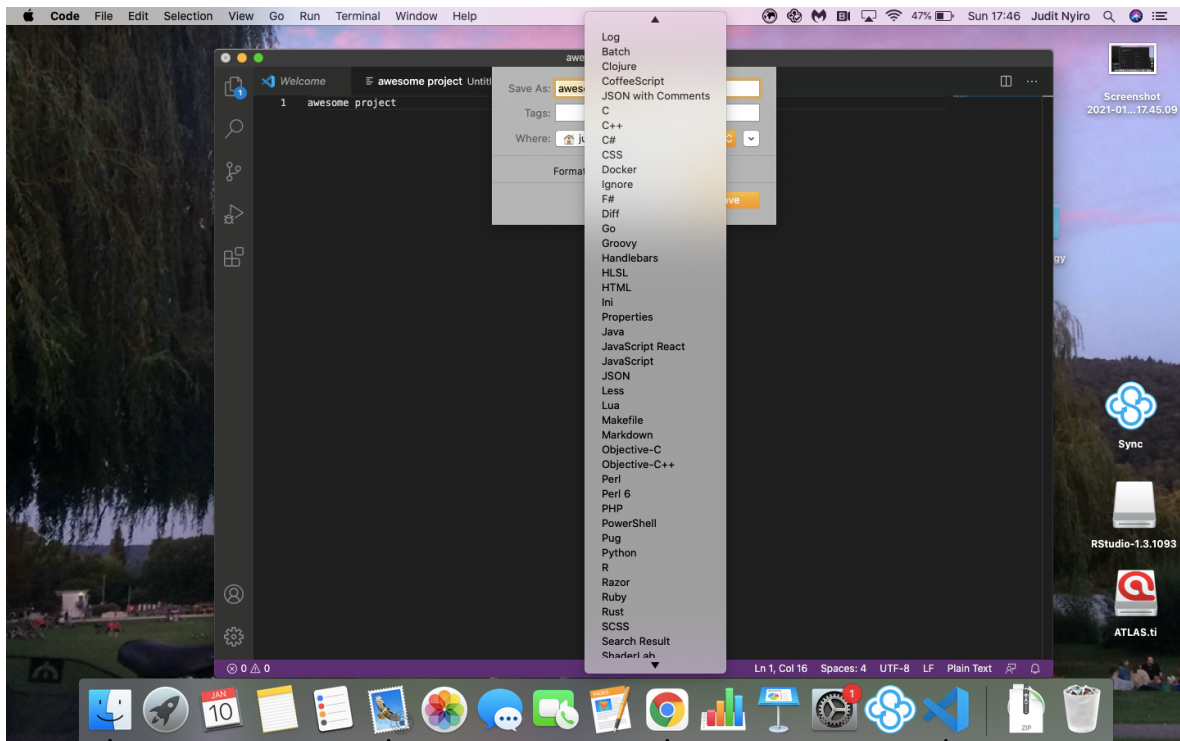
This is what the basic interface looks like. For now, you do not need to know any of its complex functions, just be able to use it for viewing and basic editing.



MAC USERS: for some reason Notepad++ does not run on MacOS. Here are some alternatives for you:

- Visual Studio: <https://code.visualstudio.com/docs/?dv=osx>
- Atom: <https://atom.io/>
- Coda: <https://panic.com/coda/>

This is what the Visual Studio interface looks like:



Great, you are all set in terms of basics!

## 16.3 Data Preparation

### 16.3.1 Take a closer look at your directory

Open your semi-personalized Empty ROCK directory, which should now have an R project file customized according to your preferences. Please feel free to add any directories you already know you will need during the project, e.g. “conceptualization”, “operationalization”, “code development”, “manuscripts”.

Now click on the Data folder. You will see that it already contains sub-folders and those contain files.

```

data
results
scripts
.gitignore
.gitkeep
.gitlab-ci.yml
.Rhistory
OncoComm_ena
README.md

```

The term “sources” signifies the files you plan to code (or have already coded), and by “code” here we mean qualitative (inductive or deductive) coding of text files for research purposes. The “Raw

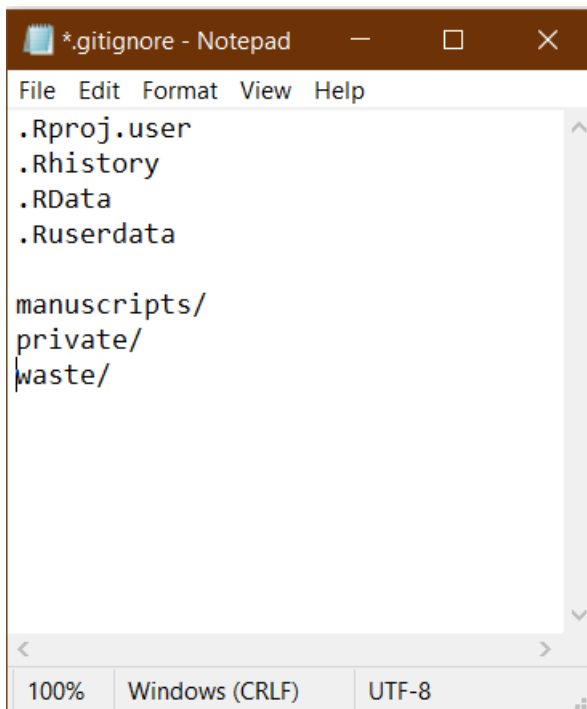
`sources`” folder in the directory “Data” is to house your raw data, e.g. interview transcripts, focus group transcripts, transcribed audio diaries.

```

├── .gitkeep
├── _PRIVATE_metropolis-a
├── gloria-a
└── metropolis-b

```

You will notice that we have put some sample text files here for you. Feel free to delete those as you replace the folder content with your own raw data (see step: “Place your raw data into the appropriate directory”). You will also see a file called “`.gitkeep`”; each folder in your main directory (including the main directory) can be told to sync to your Git repository or not. When you see a file called “`.gitkeep`” in a folder, it means that the contents of that folder will always sync to Gitlab when you “push” in RStudio (see: xx). You may not want everything on your PC to sync to Gitlab, though. If you do not want a folder to sync to your repository, then open the file called “`.gitignore`” in the main directory and add the name of the folder you wish to disable from sync-ing.

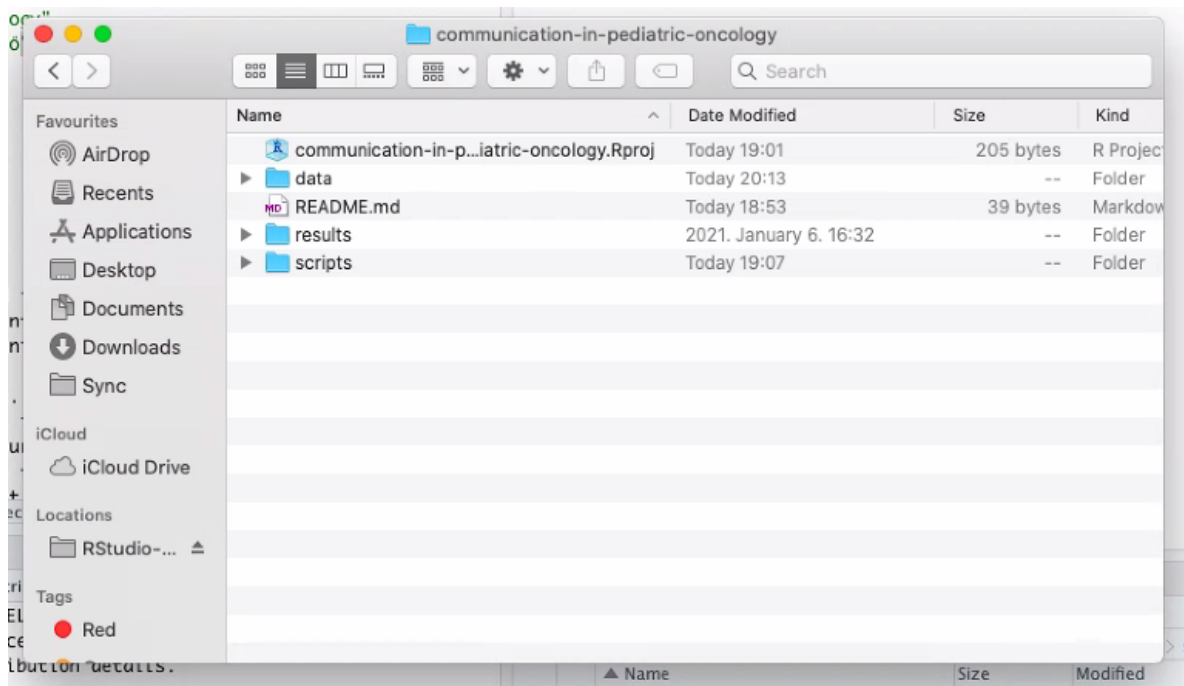


Add the folder name and then a backslash to indicate that Git should ignore the entire folder. Here we are telling Git to not sync folders called “`manuscripts`”, “`private`”, and “`waste`”. This way, even when we have set our Gitlab repository visibility level to “Public”, these three folders will not appear in the repository at all (but you will be able to see and access them on your PC).

MAC USERS: Some Mac users may not see the files with a `.git` extension; don’t worry, it’s still there. You can either download a piece of software that helps your OS see these files, or use RStudio to handle them. If you look at your directory in RStudio, you should be able to see `.git` files as well.

Example of directory contents on MacOS:

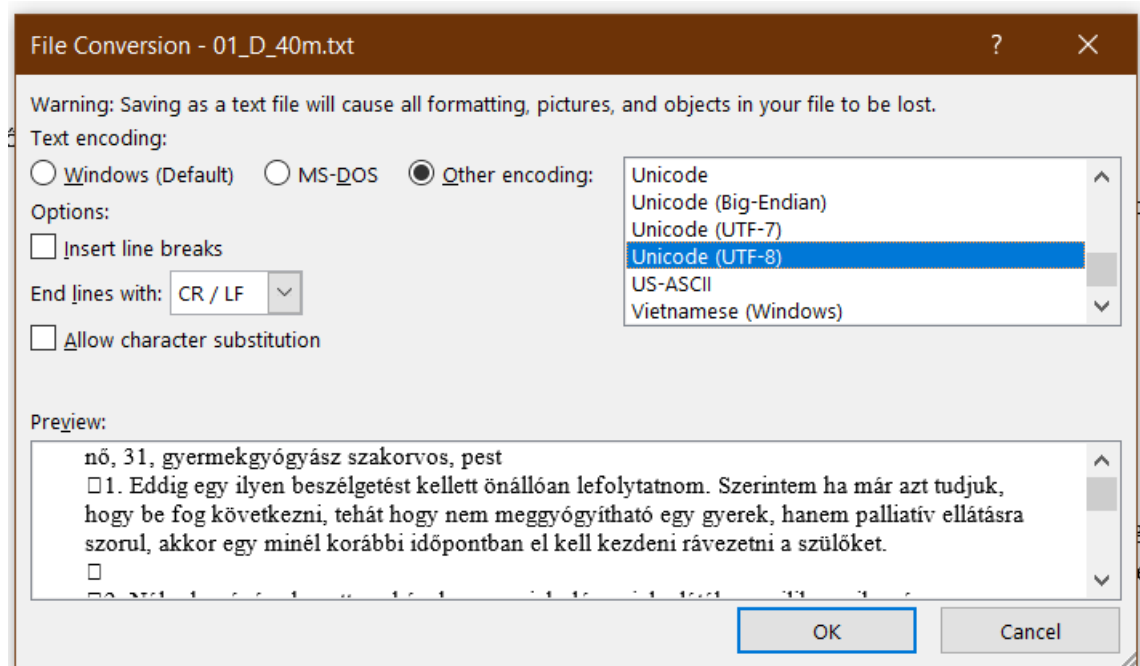




### 16.3.2 Place your raw data into the appropriate directory

Replace the files in “Raw sources” with the files that constitute the sources in your project. Make sure to save these as files with the extension .txt

If you have your transcripts in Word, then you can just “Save as” a plain text file. As you convert, depending on the language of the transcript, you might have to change the encoding. Choose “Unicode (UTF-8)” in the “Other encoding” section. After double-clicking on that, you should be good to go.



You will have the option to code your sources manually (i.e. code texts by hand or with iROCK) or

autocode them (automated coding with ROCK), note that the corresponding folders are already set up for you. Automated coding may have several rounds, the directory structure is set up so that coded sources from multiple rounds can be placed into separate directories. See section “xx” for more on autocoding.

- 01--raw-sources
- 02--cleaned-sources
- 10--sources-with-UIDs
- 11--autocoded-sources--round-01
- 12--autocoded-sources--round-02
- 20--manually-coded-sources
- 21--recoded-sources--round-01
- 21--recoded-sources--round-02
- .gitkeep
- attributes.rock

Please note that a file called “**attributes**” is also present in your “Data” folder. This file will contain your metadata and attributes, characteristics about your sources (files to code, or coded files), your cases (data providers), or any other aspect of your data that is relevant to your research project. See step “Designate your attributes” for more on attributes.

### 16.3.3 Designate your attributes

Open the file called “**attributes.rock**” in the “Data” directory with Notepad++ (right click on file and “Edit with Notepad++”).

```

1 # This file contains attributes. Attributes don't all have to be specified in the same file; it's
2
3 ---
4 ROCK_attributes:
5 -
6     caseId: 1
7     artistName: "Dream Theater"
8     songName: "Metropolis Part 1: The Miracle And The Sleeper"
9     year: 1996
10 - caseId: 2
11     artistName: "The Dear Hunter"
12     songName: "Gloria"
13     year: 2008
14 ---
15

```

The top line explains that you can have your attributes listed in the source they pertain to or have them all in one file. We will go through the latter version here. (Please note that because the top line begins with a hashtag, it is regarded as a comment in R and thus can stay in the document without interfering with any future operations.)

After you have compiled the attributes (or metadata) you wish to include in your analyses, you will need to convert that information so that the **rock** package can understand it. For this conversion, it is crucial that you keep the formatting you see in the “**attributes.rock**” document you have opened.

You can see that the format begins with three dashes, then a line break, the expression “**ROCK\_attributes:**”, then a line break. Next, we have one dash after two spaces; in the next line we begin the actual list of attributes, which, in the example are: **caseId**, **artistName**, **songName**, and **year**. Each attribute is preceded by four spaces. Two spaces followed by a dash separate cases, and after the last case, the list of attributes is finalized with three dashes.

Note that the values for **artistName** and **songName** are in quotation marks, while numbers (e.g. **year**) are not. Please keep this format, while entering your own attributes and their values. Also, please remember that attributes must correspond across cases if you want to aggregate and compare them.

Here is another example with the same attributes but different values:

<pre> --- ROCK_attributes: -   caseId: 1   artistName: "Madcon"   songName: "Beggin"   year: 2007 -   caseId: 2   artistName: "U2"   songName: "One"   year: 1992 --- </pre>	<pre> ---¶ ROCK_attributes:¶   ·-·¶     ···caseId:·1¶     ···artistName:·"Madcon"¶     ···songName:·"Beggin"¶     ···year:·2007¶   ·-·¶     ···caseId:·2¶     ···artistName:·"U2"¶     ···songName:·"One"¶     ···year:·1992¶ ---¶ </pre>
--	---

The dash separating cases can receive a line to itself or be on the same line as the subsequent case. The picture on the right shows you characters that are usually not displayed but are there nonetheless (spaces and line breaks) to help you see the formatting in entirety.

### 16.3.4 Designate your persistent IDs

In this step, we will identify persistent IDs in your project and let ROCK know about them. Persistent identifiers are codes that repeat every utterance until told otherwise: once they are applied to an utterance in a source, the rock package will automatically apply them to all subsequent utterances in that source. A good example of a persistent ID is “case”, which for an interview conducted with a single participant would persist over the entire transcript, or for focus groups, case IDs would indicate when a speaker changes.

Let’s take the example of working with individual interviews; we want to indicate for every utterance in the transcript that it was spoken by the participant. Instead of having to code every single utterance in an interview with e.g. “caseID: 2”, you can tell ROCK to consider this code as repeating up until when it encounters another such code. You may have several cases contributing to one source, such as in a focus group discussion. Persistent IDs in this instance would be useful as well, because you only need to indicate the participant (caseID) when there is a change in speaker.

Most commonly, persistent IDs will be caseIDs and various forms of mid-level segmentation.

Open the R project and specifically, your script. Locate the section called “Configuring the ROCK package” (here, i.e. in the figure below, on line 114-125).

```

113 -
114 - ```{r}
115 -
116 - ### Set the non-default ROCK options
117 - rock::opts$set(
118 -   persistentIds = c("caseId",
119 -                     "coderId"),
120 -   sectionRegexes = c(paragraphBreak = "---<<paragraph_break>>---",
121 -                      topicListSwitch = "---<<topiclist_switch>>---"),
122 -   silent = TRUE
123 - );
124 -
125 - ```
126 -

```

Locate the code in the grey box (here: beginning on line 92), and while keeping the format as is, switch the items you wish to include and delete the items you wish to exclude.

In this example, we have designated “caseId” and “coderId” (lines 96-97) as persistent IDs because we have one case per source (e.g. one participant per interview) and we have multiple raters and wanted to keep track of who is coding which source.

We have also designated two forms of segmentation (lines 98-99), one is a way to indicate when a paragraph ends and another begins (paragraph-break), and the other form of segmentation is for indicating a change in topic (topiclist-switch). According to the ROCK standard, your segmentation labels should follow this format:

```
---<<your_chosen_label>>---
```

Don’t forget to keep the symbols on either side of the actual label because those symbols are needed to let ROCK know this is a form of segmentation. To provide a full example of how to switch your segmentation labels in the code:

```
91
92 - ```{r}
93
94 ### Set the non-default ROCK options
95 rock::opts$set(
96   persistentIds = c("caseId"),
97   sectionRegexes = c(TopicSwitch = "---<<topic_switch>>---"),
98   silent = TRUE
99 );
100
101 - ```
102
```

Note that here we have reduced persistent IDs and section regexes to one item each. To do this, all we did was take out from the parentheses what we did not need, making sure to keep the parentheses themselves and the commas at the end of the lines originally containing commas. After you have made modifications and have run this command, you should see this in the console:

```
>
> ### Set the non-default ROCK options
> rock::opts$set(
+   persistentIds = c("caseId"),
+   sectionRegexes = c(TopicSwitch = "---<<topic_switch>>---"),
+   silent = TRUE
+ );
>
```

### 16.3.5 Clean your sources

Sources are usually messy, i.e. they are not neatly transcribed and necessitate formatting. The main purpose of “cleaning” in the **rock** package is making sure that utterances are split along “utterance markers”. Utterances are the smallest meaningful unit of data in your study, coding occurs on this level. Utterances are separated by utterance markers; the default in the ROCK is a newline character (a character that indicates a new line) and the **rock** package can insert those for you.

Go to your console in RStudio and copy this command into it: `?rock::clean_sources` and hit enter. On the right you will see a list of default actions the **rock** package will perform if you use the Clean Sources command. As you can see, the description says: “Cleaning consists of two operations: splitting the source at utterance markers, and conducting search and replaces using regular expressions.” For now, we will not look at the extra search and replacements you can ask the **rock** package to do. We will discuss that here (xx).

The **rock** package is set to consider sentences as utterances and insert newline characters between all sentences. These are the actions the **rock** package will perform to clean up your data:

- Double periods (..) will be replaced with single periods (.)

- Four or more periods (... or .....) will be replaced with three periods
- Three or more newline characters will be replaced by one newline character (which will become more, if the sentence before that character marks the end of an utterance)
- All sentences will become separate utterances (in a semi-smart manner; specifically, breaks in speaking, if represented by three periods, are not considered sentence ends, whereas ellipses (“...” or unicode 2026, see the example) are.
- If there are comma’s without a space following them, a space will be inserted.

If your utterances are not defined as sentences, please see xx.

Now, locate the section titled “## Preparing and cleaning sources” in your script within your R project. Locate the command in the grey box. Make sure your sources are placed into your “01--raw-sources” directory and are in text files (files with a .txt extension). Now click on the green “play” button within the command’s grey box.

```

101 );
102
103 ~~~
104
105 ## Preparing and cleaning sources
106
107 Freshly transcribed sources are not always very neatly and consistently formatted. Therefore, some cleaning is often
108 beneficial. The main purpose of cleaning is making sure that utterances are split by the utterance marker (the ROCK
109 default is a newline character, '\n', which most operating systems render as a new line).
110
111 By default, the 'rock' package tries to smartly insert newline characters between all sentences (see
112 'rock::clean_sources' for more details). You can also use this to do replacements before or after the insertion of
113 the utterance markers. For example, if the interviewed participants live in six cities, say Amsterdam, Beijing,
114 Canberra, Dhaka, Edinburgh, and Freetown, and have relatively rare afflictions, to preserve their anonymity, all city
115 names can be replaced with the text "GEOGRAPHICAL_REFERENCE" by combining the city names into regular expression
116 "^(Amsterdam$|Beijing$|Canberra$|Dhaka$|Edinburgh$|Freetown$)" and specifying both in the
117 "extraReplacementsPre" argument, as done in this example.
118
119 ~~~{r}
120
121 rock::clean_sources(
122   input = here::here("data",
123     "01--raw-sources"),
124   output = here::here("data",
125     "02--cleaned-sources"),
126   extraReplacementsPre =
127     list(
128       c("^(Amsterdam$|Beijing$|Canberra$|Dhaka$|Edinburgh$|Freetown$)",
129         "GEOGRAPHICAL_REFERENCE")
130     )
131 );
132
133 ~~~
134
135 Instead of using a regular expression, you could also specify six pairs to search and replace: one for each city. In
136 this example, regular expressions are efficient, but not necessary. In many cases, however, regular expressions can
137 make one's life considerably easier. To learn more about regular expressions, see the [Psy ops
138 Guide](https://psy-ops.com/regexes).
139
140 Note that if you don't want to clean, or if you have already cleaned your sources, you can always do any replacements
141 using the 'rock::search_and_replace_in_source()' and 'rock::search_and_replace_in_sources()' commands.
142
143 ~~~
144
145 Organizing the files
146
147 R Markdown
148
149 Console
150
151 D:/GIT projects/OncoComm_ena/
152
153 type 'license()' or 'licence()' for distribution details.
154
155 Natural language support but running in an English locale
156
157 R is a collaborative project with many contributors.
158 Type 'contributors()' for more information and
159 'citation()' on how to cite R or R packages in publications.
160
161 Type 'demo()' for some demos, 'help()' for on-line help, or
162 'help.start()' for an HTML browser interface to help.
163 Type 'q()' to quit R.
164
165 > |

```

This action will place your cleaned sources into the folder called “02--cleaned-sources”. Open that directory and check your sources to make sure that each sentence is on a separate line. You may

also notice that the `rock` package has changed the extension of your files to `.rock` to make them recognizable. They remain plain-text files, and so will still be editable with your text editor.

### 16.3.6 Add UIDs to your sources

Now that your utterances have received a separate line in your sources, the next step is to give each utterance a unique identifier. This identifier will serve to “collapse” all your codes and attributes onto when you aggregate your data (see: xx). Utterance identifiers clarify which utterances (e.g. sentences) belong to which case (e.g. participant), what attributes those cases have, and what discourse codes utterances were given. This enables you to group your coded utterances according to participant (or any of their attributes) and investigate code co-occurrences on the level of utterance or higher levels of segmentation, such as stanza (i.e. sets of utterances).

Open RStudio and locate the section “**## Prepending utterance identifiers (UIDs)**” in your script within your R project. The command for prepending unique utterance identifiers (uids) is in the grey box under this section heading. Running this command will ask the ROCK to take the files in folder “02--cleaned-sources”, add uids to each line (e.g. sentence), and put the files with uids into the directory “10--sources-with-UIDs”. Note, these files will have the same exact text, with the only difference that each line will receive a combination of letters and numbers added to the front of the line. This unique combination will be within two square brackets:

```
[[uid=7b9phj1x]] V: Igen.
[[uid=7b9phj1y]]
[[uid=7b9phj1z]] K: Mesélnél ezekről egy kicsit, hogy micsoda és mikor kezdődött?
[[uid=7b9phj20]]
[[uid=7b9phj21]] V: Na, hát amire, szóval, hogy ez nekem gyerekkorom óta van valójában a, tehát ami biztos, hogy van az székrekedés.
[[uid=7b9phj22]] Csecsemőkoromról annyit tudok, hogy hasmenes gyerek voltam, de sokan vagyunk ilyenek és ilyen nagyon korai gyerek élményem az, hogy beöntéseket kapok.
[[uid=7b9phj23]] Van, hogy az apámék ott imádkoznak, hogy ott ülök már, tehát annyira nagy vagyok, hogy tudok már a nagy wc-n ülni és hogy így ketten is vannak körülöttem.
[[uid=7b9phj24]] Nem tudom, a nagymamám meg az apám, hogy így drukkoljanak, tudod.
[[uid=7b9phj25]] Meg, hogy szegény gyerek, így full stressz.
[[uid=7b9phj26]] Borzalmas.
```

Congratulations, your sources are now ready to be coded!

## 16.4 Coding and Segmentation

### 16.4.1 Designate your codes and section breaks

This section is for researchers coding their data deductively (using predetermined codes for coding); if you are working inductively, please see section xx. We suggest that you use iROCK to perform coding; this is an interface that allows you to upload the sources you want to code (and segment), as well as your codes (and section breaks), and just drag and drop them into the document. Please see the next section on how to use iROCK.

If you are ready with your codes, you will need to make a separate text file that contains the codes according to the ROCK standard. This format is the following:

```
[[your_code_identifier_here]]
```

If you are working with hierarchical codes, you have to use the hierarchy marker to indicate a code’s parent codes, e.g.:

```
[[Info>source>media]]
```

Here, the parent code is “Information”, the child code is “source” and the grandchild code (what we will actually use in coding) is called “media”. Below is an illustration for a list of codes in a text file:

```
[[I>s>net]]
[[I>s>fam]]
[[I>s>pers]]
[[I>s>peer]]
[[I>s>self-help]]
[[I>s>media]]
[[I>s>cam]]
[[I>s>md]]

[[I>j>intu]]
[[I>j>sci]]
[[I>j>time]]
[[I>j>miscel]]
```

Please note that only these characters will be considered a code, provided they fall between two square brackets: `[a-zA-Z0-9_>]`. Any other sequence of characters, regardless of whether it's between `[]`, will not be seen as a code identifier. Also, make sure that none of your low-level code labels are identical, as even if they belong to different parents, the ROCK will not be able to parse them as different codes.

Several raters might be coding the same corpus with different codes in your project. In this case, we suggest creating separate text files for each code cluster (or parent code) so that the code list only contains codes that a certain rater would use. Of course, if multiple raters are using the same codes, then the text document should be shared among them.

If you also intend to segment your transcripts (e.g. according to question-response or topic), you should create a text document with your section breaks as well. Here is an illustration of three section breaks in one file:

```
---<<stanza_delimiter>>---
---<<topic_delimiter>>---
---<<paragraph_delimiter>>---
```

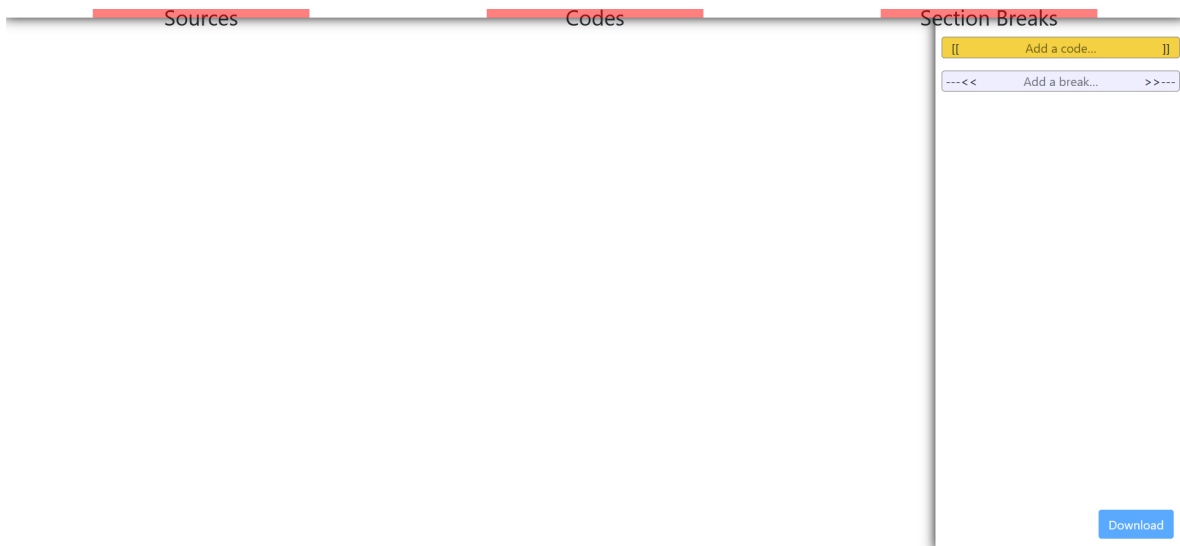
Do not use hyphens in section break labels! As with code labels, they should consist of these characters: `[a-zA-Z0-9_>]`.

Once you have created these documents (which can be placed e.g. in their own separate folder), you can begin coding your sources! We suggest you use iROCK for this purpose, see the next step for details!

## 16.4.2 Using iROCK to code your sources

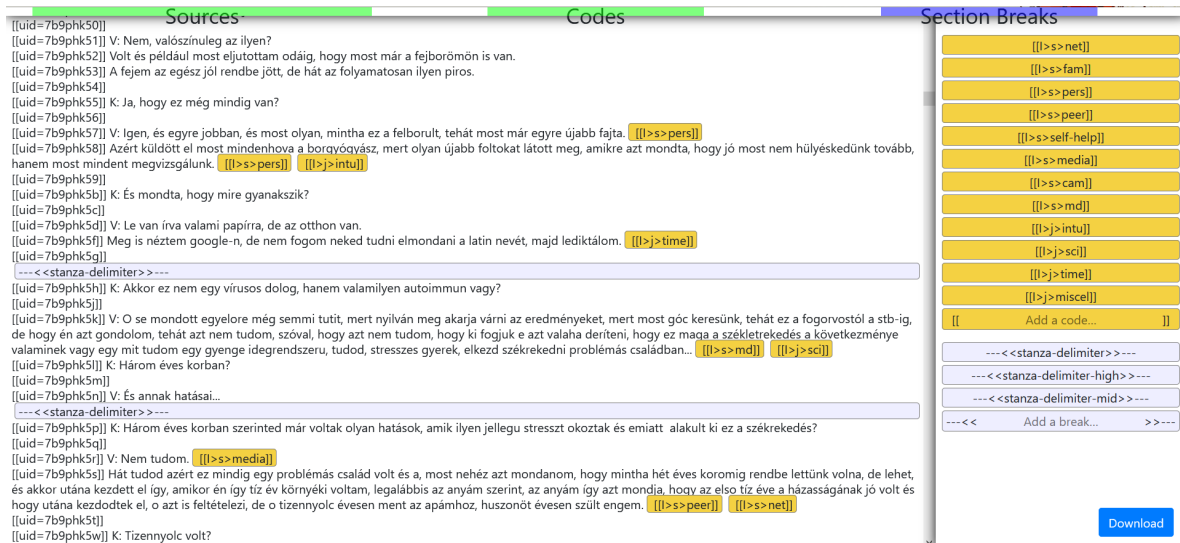
This section is dedicated to familiarizing you with the interface for ROCK, or iROCK, which was created to ease manual coding of data (if you want to automate your coding process, please see section xx). First, let's go through the preparations involved in coding and segmentation. Please click on this link: <https://i.rock.science> to reach the iROCK interface. This is what you will see:





Going from left to right, by clicking on the buttons at the top you will be able to upload your Sources, your Codes, and your Section breaks (provided you are segmenting your data). The buttons on the right pane allow you to add your codes and section breaks inductively by typing in their label and hitting enter.

If you are using deductive coding, we suggest you upload your text file containing your codes, because it may be tedious to type in the list every time you begin coding a new source. Your codes will appear in the pane on the right and you will be able to drag and drop them on the utterance you want to code (i.e. tag an utterance with a code). Sections breaks work in a similar manner, except those are to be positioned in between two utterances (i.e. slice your text into fragments). When you have uploaded these components and started coding, the interface will look something like this:



Please note that segmentation cannot be nested at this point in the ROCK, that is, a section break operates as a persistent identifier: it repeats at each utterance until told otherwise. In the figure above, there are three text fragments delimited by two “stanza delimiters”; thus, every utterance before the first delimiter will be labeled with “0”, every utterance between the first and second delimiter will be labeled with “1”, and all utterances after the second delimiter will be labeled with “2” until a new section break is added to the text. We are working on a way to create threaded data, which can be employed as nested segmentation also.

Multiple section breaks ARE allowed on one line.

When you have finished coding, just click on the “Download” button in the lower right corner and place the coded source into your “data” directory, within the “20--manually-coded-sources” folder.

If you want to work on a source iteratively, or you want to pass it on to another rater and want them to be able to see your coded version, you can download the (partially) coded source and then just upload it next time you want to work on it. When you download a file from iROCK, it saves all information in a text file with the .rock extension. This file can be re-loaded into iROCK at any time and you can manually modify the file in any text editor.

### 16.4.3 Using the rock package to autocode your sources

Tutorial section coming soon!

### 16.4.4 Add identifiers to your sources

Often, qualitative sources have a set of characteristics that are independent of coding. In the ROCK, these are called “attributes”, and they are designated to sources using “identifiers”. An identifier is a special code that can be used to identify sources. For example, identifiers can be used to code which participant was interviewed; who the interviewer was; what the location of the interviews was, or whether the interview was conducted during the morning, afternoon, or evening.

These identifiers can then be used to associate attributes to utterances. By default, the ROCK uses the identifier “caseId” to designate data providers. It is configured as a so-called “persistent” identifier, which means that it does not only apply to the coded utterance, but is automatically applied to all following utterances until another “caseId” identifier is encountered.

These identifiers are ideally added during or just after transcription, but they can also be added around the time of (manual) coding. The default ROCK case identifier format is:

```
[[cid=EnterLabelHere]]
```

You can use e.g., numbers or pseudonyms (of course, never use participants’ real names!). Here are a couple of examples:

```
[[cid=25]]
[[cid=Patient_7]]
[[cid=Larry1997]]
```

When you have created a caseID for all your participants, you still need to let the ROCK know which case contributed to which source. The caseID of your data provider should be placed into every source they contributed to. If only one case contributed to a source, you can just copy-paste the cid into the (coded) text file to the top of the document. If your sources contain utterances from more than one participant...

more coming soon!

## 16.5 Aggregation and Analyses

## Chapter 17

# Using the ROCK for Epistemic Network Analysis

Epistemic Network Analysis (ENA) is a convention and software housed within the larger methodological framework of Quantitative Ethnography (Shaffer, 2017). ENA was originally developed for *modelling and comparing the structure of connections among various elements in a data set*. In case of qualitative data (narratives), connections in the data are generated from co-occurrences of codes within segments; these co-occurrences are visualized in a network. ENA is a useful tool if one is working with *(a large number of) variables in a single system* and can benefit from modelling complex structures in search of patterns in the data. To read more about ENA and Quantitative Ethnography, see Shaffer (2017).

Below we offer guidelines for using the ROCK to prepare data for use in ENA. *the ROCK provides help in the process of preparing and performing coding and segmentation, merging coded documents from multiple raters, and creating the qualitative data table (CSV file) necessary for making networks*. the ROCK will aid you the most if you are working with continuous narratives (e.g. semi-structured interviews, from more details see below: Planning Segmentation) and performing manual coding (as opposed to automated coding, for more see NCoder).

The following is a *step-by-step account of how to employ the ROCK in creating networks from your data*, but these steps are not in strict order as work processes are highly dependent on the project in question. the ROCK conventions will be illustrated with a worked example. In general, the guidelines will be structured as follows: theoretical considerations presented in separate sections, instructions for use in the ROCK, and the corresponding information from our worked example under each sub-section.

### 17.1 Starting point

These guidelines assume that the researcher is familiar with the basic tenets of Quantitative Ethnography (QE) and ENA (although important terms will be clarified). The starting point of the guidelines also presupposes that the researcher is working with an *anonymized database of raw, qualitative data* that was collected in a systematic manner during a project where the research question, sub-questions, methods, and sampling have all been established. The guidelines do not provide advice on research design.

As a preliminary step, please install the required software as explained in sections 12.1 and 12.2 in Chapter 12.

### Our example

Very succinctly, we were interested in modelling cognitive and behavioral patterns in patient decision-making processes regarding choice of therapy, i.e. for a certain diagnosis, what sources of information are considered, what specific decisions are made during the patient journey, and what conceptual framework the patient has concerning illness causation. We wanted to know what cognitive patterns underlie the decision to use different types of medicine (conventional and non-conventional). To read more about the research questions and design (methods, sampling, etc.), please see the methods section of @zorgo\_patient\_2018 and @zorgo\_qualitative\_2018, and the methodological considerations in @zorgo\_epistemic\_2019.

## 17.2 Planning coding

### 17.2.1 What is a code?

One aim of QE is to localize patterns within a community of practice (culture or subculture), which may be referred to as “Discourse” (capitalization intended). To do this, the researcher gathers “discourse”, that is, data from the scrutinized community, such as transcripts from interviews or focus groups, field notes from observations, etc. “Codes” (capitalization intended) can be defined as culturally relevant and meaningful aspects of a Discourse, the elements that the researcher wishes to address in the process of analysis; these elements will constitute the nodes of the network model. Finally, “codes” are manifestations of these elements that one identifies in their data, i.e. evidence for Codes within the narratives. For a more elaborate description of the QE framework see: xxx.

### 17.2.2 Types of coding

As with coding qualitative data in general, there are several decisions one needs to make. Should I code with a predetermined set of codes (deductive coding) or should I allow for the codes to emerge from the narratives as I progress in analysis (inductive coding)? Both manners of coding have their advantages and disadvantages (for more details see Smith & Osborn (2008), Denzin & Lincoln (2000) and Babbie (2007)), the ROCK enables the researcher to employ one or the other, or even both.

Another consideration, with both deductive and inductive coding, is whether the set of codes should be hierarchical or not. A hierarchy would imply that some codes constitute part of other, more abstract codes, such as the parent code **fruit** containing the child code **banana**. If codes are not arranged hierarchically, they would still constitute a single analytical system in light of the research question, but cannot be conceptualized as containing one another, such as **fruit**, **dairy**, **meat**, **grains**, and **vegetables**. Again, the ROCK supports both hierarchical and non-hierarchical constructs.

### 17.2.3 How to represent codes in the ROCK

The general format for representing codes in the ROCK is placing the code name (e.g. fruit) in between two square brackets, for example: `[[fruit]]`. If the code name contains two or more words, we suggest using an underscore to separate them, e.g.: `[[exotic_fruit]]`; we also suggest keeping the code names concise but informative.

Both hierarchical and non-hierarchical inductive coding can be generated in the above format with the help of the Interface for the ROCK (the iROCK) platform (see below: Coding and Segmentation).

Both inductive and deductive hierarchical codes necessitate a greater-than sign to signal their place in the overall structure, for example: `[[fruit>banana]]` connotes a two-level hierarchy; `[[fruit>exotic>banana]]` connotes three levels.

Hierarchical and non-hierarchical deductive codes need to be specified before coding begins and listed in a file designated specifically for this. Deductive codes may be structured in several code clusters or trees (for more detail see: xxx). Non-hierarchical deductive coding should follow the above format for the ROCK codes, for example, the codes in the previous example would look like this:

```
[[fruit]]
[[dairy]]
[[meat]]
[[grains]]
[[vegetables]]
```

Hierarchical and non-hierarchical deductive codes are essentially a list of codes, in the above format, placed into a separate file, preferably with a `.rock` extension (see previous section: General background and introduction: `.rock` file format).

### Our example

In our case, *Discourse* refers to patterns in cognition and behavior among patients using biomedicine only, and patients using non-conventional medicine to treat their illness(es). We can also consider the individuals in these two groups as all belonging to larger groups delimited by their “primary diagnosis”. In our case, we began by choosing four diagnoses (D1-4): Diabetes (D1), Musculoskeletal diseases (D2), Digestive diseases (D3), and Nervous system diseases (D4). Thus, every individual in our study belongs to a group indicating their primary diagnosis (D1-4) and their choice of therapy. Individuals can be grouped further based on other characteristics (for more on this subject, see below: Designating Attributes).

Our *discourse* consists of transcripts from semi-structured interviews conducted with patients belonging to one of the four types of diagnosis groups and representing different choices of therapy (for more on the latter, see below: Designating Attributes).

The *Codes* we were interested in encompass the three main areas of interest within the project: sources of information (epistemology), concepts of illness causation (ontology), and decisions in the patient journey (behavior). We employed both deductive and inductive coding. We coded the above three areas of interest with a predetermined set of hierarchically organized codes, on three levels of abstraction, comprising 52 low-level *codes* in total. The complete code tree can be accessed here: xxx. Our inductive coding only concerned illnesses. As interviewees also spoke about comorbidities during the interview, we found it important to distinguish among primary diagnosis and other, specific comorbidities the patient is referring to within the narrative.

## 17.3 Planning Segmentation

### 17.3.1 What is discourse segmentation?

Segmentation, according to QE, is the process of dividing data up into sensible structures, meaningful parts. There are different levels and modes in which one can segment narratives; these segments will be important in the creation of a network because connections are formed based on the number of code co-occurrences within the designated segments.

Following the QE framework, there are three important levels of segmentation to consider: the smallest unit of segmentation (utterance), a middle level (stanza), and a high level (unit). At this point we will address the first two of these, units will be dealt with later (see below: Creating Networks).

An utterance is the smallest entity of analysis in a narrative. This can be one sentence (e.g. a semi-structured interview’s utterances are sentences articulated by the interviewee) or more than one sentence (e.g. one remark made by one participant in a focus group). An utterance can also be one line or one entry in a field journal, for example. In any case, coding will occur at this level.

Albeit coding occurs on the level of utterances, co-occurrences are computed based on a higher level of segmentation, the stanza. A stanza is a level of discourse structure composed of one or more utterances that occur in close proximity and discuss the same topic (i.e. recent temporal context). Stanza size reflects how much content the researchers consider indicative of psychological proximity. Researchers who are only interested in tightly connected concepts may prefer shorter stanzas, however, if the research topic concerns broader, more complex issues, researchers may want to define larger stanza sizes. Stanza size crucially determines analysis results, thus the rules for segmentation should not be arbitrary and should be made transparent. In order to explore various versions of segmentation, more than one stanza-type can be employed (i.e.: multiple ways of defining stanza length and multiple identifiers). Furthermore, stanzas constitute merely one way of segmenting data on the middle level, one may want to utilize many forms of section breaks (for details see: Cognitive Interviews).

### 17.3.2 Continuous and discontinuous narratives

Qualitative data can come in many forms and have varying characteristics, an anthropological field journal presents us with very different text compared to a focus group or an interview, for example. Thus far ENA has mainly been used for discontinuous data, namely, teams of people performing tasks in a common virtual reality or performing virtual tasks in a shared physical reality (see: Ruis et al. (2018)). Similar to focus group situations, these studies worked with data that was supplied by several participants and on several, discrete occasions. Naturally occurring “turns of talk” among participants provide for excellent segmentation, for example, students discussing how to accomplish a common task in a chatroom (Bressler et al., 2019).

Continuous narratives are distinguished from discontinuous narratives by the lack of naturally occurring possibilities for segmentation; such text may originate from the transcript of a semi-structured interview or an audio diary. Because there are no “turns of talk” (or they are between interviewer and interviewee and yield little contextual information), and the whole text may be intricately connected internally, demarcating stanzas becomes a challenge. Similar problems may occur with the smallest unit of analysis, especially if it is defined as “one sentence”. The verbatim transcription of spoken speech comes with inherent subjectivities; as a sentence in speech may persist across vast reaches, the transcriber makes many judgement calls in punctuation. Yet, as co-occurrences are computed based on stanza, length of utterance is not decisive in this particular case.

### 17.3.3 How to represent segmentation in the ROCK

Utterances are represented in the ROCK by something called an “utterance identifier” (UID). It is one line in the ROCK file (a line being defined as zero or more characters ending with a line ending). When the ROCK reads a certain file containing text and utterance identifiers, it splits each file at the line endings (with newline characters). This parsing is necessary to perform the coding of each utterance in a file and be able to work with that information later on. An example of a UID is: `[[uid=73ntnx8n]]`, each utterance receives a unique identifier. Utterances may be grouped together with the aid of section breaks, one of which is the stanza; the ROCK uses this particular format: `<<stanza-delimiter>>`, where the name “stanza delimiter” may be changed according to the segmentation needs of the project.

#### Our example

In our project, an *utterance* is defined as one sentence. Each sentence constitutes one line in the ROCK; each utterance/line receives a unique identifier. Regarding the *stanza*, we employed the generic defi-

inition above, but we had three different raters perform segmentation autonomously based on their judgement of psychological proximity. Their three identifiers were: `<<stanza-delimiter-low>>`, `<<stanza-delimiter-mid>>`, and `<<stanza-delimiter-high>>`. The names indicate the level of knowledge each rater had concerning the scrutinized research topic; “low” signified a (naïve) rater not connected to the research project, only privy to the interview transcripts. “Mid” was employed by a research assistant with a significant amount of prior knowledge on research objectives and codes, while the “high” delimiter was used by the principle investigator. Thus, we had three different stanza-types for all interview transcripts in order to explore which stanza-type creates the best models.

## 17.4 Designating Sources and Cases

### 17.4.1 What is a source and how is it represented in the ROCK?

A source is a file with content to code (or coded content); it can contain the transcript of an interview or a focus group discussion, or even a list of twitter posts. Sources comprise one or more utterances from one or more participants of a study. Sources should be plain-text files and can bear any name, although they should be kept concise, as these will be displayed in the ENA interface later on. Information relevant to the study can also be displayed in the name, such as: “5-female-30s”, indicating this is the fifth interview and it is with a female participant in her 30s.

### 17.4.2 What is a case and how is it represented in the ROCK?

A case signifies a participant, a provider of data within a study. This can be a person, a family, an organization, or any other unit of research. In case of individual interviews, the source and the case may be identical, but it is important to distinguish between these as one source can contain data from many cases. For example, a focus group transcript constitutes a source, while the six participants connote separate cases within the source. Each case receives a unique identifier (case id, CID) and is represented in the ROCK with two square brackets and a designated name, for example: `[[cid=alice]]`. Naturally, in anonymized studies it is preferred to have an alias of some sort, it can even be a number.

## 17.5 Designating Attributes

### 17.5.1 What is an attribute?

Cases can be supplemented with characteristics or variables; we refer to these as “attributes” (ENA term: metadata). For each participant you may want to collect additional data, such as demographic variables, or even conduct a survey in addition to an interview, for example, and record the answers respondents provide. You may want to register aspects, such as the date the interview was conducted, the researcher who led the focus group, or the sequence audio diaries were recorded in. Attributes essentially allow you to group participants in various ways (thus creating different networks) and enable other types of analysis through flexibly changing the sets of data you want to see a network for (i.e. conditional exchangeability), for more see below: Creating Networks.

### 17.5.2 How to record attributes in the ROCK

There are two main ways you can record attributes for cases in your study using the ROCK. One is to place this information into the source directly. For example, you open the plain-text file of your

semi-structured interview and enter the attributes above or below narrative. The other option is to create a separate .rock file containing the attributes of all participants. In either case, the format for entering attribute-related information is the following:

```

---
ROCK_attributes:
-
  caseId: 1
  sex: female
  age: 50s
-
  caseId: 2
  sex: male
  age: 30s
---
```

The above displays the aggregated version of recording attributes (illustrated with two cases). The list of entries begins with three dashes, followed by the attributes listed in the manner displayed, and ends with three dashes. In later phases of data preparation, the ROCK will read this information and assign it to the appropriate case.

### Our example

For each interview we recorded the following *attributes*: interview date, interviewer ID, interviewee ID, interviewee sex, age, and level of education, diagnosis type (D1-4), specific illness, comorbidities, illness onset, time of diagnosis, and therapy choice (treatment type concerning primary diagnosis: biomedicine only, complementary use of non-conventional medicine, alternative use of non-conventional medicine). For complementary and alternative medicine (CAM) users we also registered type of CAM use, attendance in CAM-related courses, disclosure of CAM use to conventional physician and the employed CAM modalities. For users of solely biomedicine, reason for rejecting CAM was also coded (deductively).

To summarize, here is a list of terms we have discussed thus far and some examples for each term:

Term	Explanation	Example
Discourse	Patterns within a community of practice (culture or subculture)	E.g.: patterns in cognition and behavior among those using biomedicine only, and and those using non-conventional medicine to treat their illness(es)
discourse	Data from the scrutinized community	E.g.: transcripts of semi-structured interviews conducted with patients
Code	Culturally relevant and meaningful aspects of a Discourse	E.g.: sources of information, concepts of illness causation, and decisions in the patient journey
code	Manifestations of these elements that one identifies in their data, i.e. evidence for Codes	E.g.: hierarchical, three levels of abstraction, 52 low-level, e.g. the ROCK non-hierarchical code format, e.g.: [[exotic_fruit]] the ROCK hierarchical code format, e.g.: [[fruit>banana]]



Term	Explanation	Example
Segmentation	The process of dividing data up into sensible structures, meaningful parts	<ul style="list-style-type: none"> <li>• Utterance (e.g.: one sentence)</li> <li>• the ROCK utterance identifier (UID) format: <code>[[uid=73ntnx8n]]</code></li> <li>• Stanza (e.g.: psychological proximity, recent temporal context)</li> <li>• the ROCK section break format, e.g.: <code>&lt;&lt;stanza-delimiter&gt;&gt;</code></li> </ul>
Discontinuous narratives	Text containing naturally occurring “turns of talk” among participants	E.g.: students discussing how to accomplish a common task in a chatroom
Continuous narratives	Text lacking naturally occurring possibilities for segmentation	Semi-structured interviews, audio diaries, etc.

## 17.6 Coding and Segmentation

### 17.6.1 How is coding performed with the ROCK?

Although manual coding can be performed within a qualitative data table in a spreadsheet (for more detail see: xxx), when conducting hermeneutic analysis with a high number of codes, this is unwieldy. For this reason, we developed the Interface for ROCK (iROCK), an online user platform, consisting of a file that combines HTML, CSS, and javascript to provide a rudimentary graphical user interface. Because iROCK is a standalone file, it does not need to be hosted on a server, which means that no data processing agreements are required (as per the GDPR). The iROCK interface allows raters to upload a source, a list of codes, and segmentation identifiers. Coding is performed by dragging and dropping codes upon utterances at the end of their line. Once coding is finished, the coded sources can be saved. There are a few preparatory steps you need to take before you can start coding your sources.

### 17.6.2 Creating code clusters or trees

Depending on your research design, the code structure, and the amount of codes you are working with, you may want to have separate code clusters or discrete, hierarchically organized trees. You may also want to assign different raters to specific code clusters/trees, or have several raters use the same code structure to perform autonomous coding (this allows for triangulation or even inter-rater reliability testing, for more on this subject see: xxx). In these instances, you end up with multiple coded versions of a source, for example, interview number 1 (cid=1) is coded by two raters (R1 and R2), so you end up with two coded versions of Case 1. In a situation where R1 and R2 are autonomously coding different sections of the whole code structure, they will need separate code trees or clusters. Thus, the preparation of code lists depends on how many ways the whole code structure is divided among raters: each code group (cluster or tree) should be listed in its own .rock file. Naturally, if your code structure is not divided up amongst raters then one, all-inclusive list suffices (even if that list is used by multiple raters).

### 17.6.3 Preparing sources for coding

In order to perform coding, ROCK needs to “clean” your sources, i.e. parse the plain-text files according to one sentence per line (or however utterance is defined). ROCK also needs to add UIDs to each line in order to be able to merge codes from various raters in a later phase of the process (see below:

Merging Coded Sources). Hence, you will need four directories, all of which need to be retained: original-sources; cleaned-sources; sources-with-UIDs; coded-sources. Below are the steps for preparing your data for coding:

- 1) Copy raw sources into “**original-sources**” folder (plain-text)
- 2) Locate the R chunk called “**# Preparing and cleaning sources**” and run it. This will clean the sources and save them to the “**clean-sources**” directory (and convert them to .rock format).
- 3) Locate the R chunk called “**prepend-utterance-ids**” and run it. This will load the cleaned sources, prepend the UIDs, and save them to the “sources-with-uids” directory.

You can safely repeat these steps; they will not overwrite existing files. When the files have appeared in the sources-with-UIDs folder, they are ready for coding.

### 17.6.4 Using the iROCK interface

The iROCK platform can be found at: <https://r-packages.gitlab.io/rock/iROCK/>. After you arrive to the website, you will see a ribbon at the top; by clicking on “Sources” you can upload the file you wish to code, “Codes” and “Section Breaks” can also be uploaded here. (Thus, coding and segmentation can be conducted simultaneously.) Perform coding by dragging and dropping the appropriate codes from your uploaded list to the end of each utterance. When coding is complete, download the file to your computer and place it into the “coded-sources” folder.

#### Our example

We divided up our code structure into three main areas that were reflected in the research question and the complete code tree as well (three high-level codes): epistemology, ontology, and behavior. The low-level codes belonging to these three parent codes were given to three different raters, each of whom specialized in one specific code tree. The three raters performed coding separately; none of their codes overlapped. There was a fourth rater who inductively coded the illnesses present in narratives. Each rater downloaded their coded files to a shared folder housed by a secure cloud storage (we use GDPR-compliant Sync, available at: <https://www.sync.com/>); the four raters had their own folder where they dropped every source they coded. Subsequently, the coded sources were copied to the “coded-sources” folder where all sources received a separate directory, comprising four versions of the source (three versions of coding with parent code trees and one version of inductively coded illnesses). Segmentation was performed by two of the above four researchers and one naïve rater; each used a <> matching their level of expertise (low, mid, high). Thus, in the end, *we had five versions of a source: 3 coded with parent code trees (including segmentation; high-level), 1 coded with illnesses (including segmentation; mid-level), and 1 segmented by a naïve rater (low-level)*. Because we had so many versions of one source, neither of which was complete on its own, we needed to merge these files (see below).

## 17.7 Merging Coded Sources (if necessary)

If the research design and protocol call for multiple raters coding all sources, sources need to be merged into a master document. This is necessary because the ENA interface (to be used for creating the networks) will require a Comma-Separated Values (CSV) file to be uploaded, which contains all sources, attributes, utterances, codes, and segmentation, together referred to as a “qualitative data table” with rows and columns that are ontologically consistent (Shaffer, 2017). This master document can be produced with ROCK by locating the R chunk called “**# Merging sources**” and running it. Provided the attributes were listed in a separate .rock file, use the R chunk called “**# Reading merged**

**sources**” to create the CSV file comprising the master document with attributes added. Merging coded sources may also be required if the project is later revisited with a different set of codes by the same researcher, or if many researchers are collaborating on the same project non-synchronously.

## 17.8 Creating Networks

The CSV file can be uploaded to the ENA interface (available at: <http://www.epistemicnetwork.org/>) or can be further processed with rENA (available at: <https://cran.r-project.org/web/packages/rENA/index.html>). A tutorial on how to apply the QE framework and employ ENA software can be found here: <http://www.epistemicnetwork.org/resources/>. You may benefit from reading ENA tutorials and worked examples in preliminary phases of your research, as there are other questions that need to be addressed that may, for example, influence planning discourse segmentation in your project.



## Chapter 18

# Using the ROCK for Cognitive Interviews

Cognitive interviews are a method to study cognitive validity (see Cognitive interviews, chapter 4 in this version of the book, for an introduction). The ROCK supports coding of notes or transcripts from cognitive interviews, and this chapter introduces this functionality.

There are two ROCK patterns specific to cognitive interviews:

- The unique item identifier, or `uuid` for short. To specify which item a part of the source describes, unique item identifiers can be added using the pattern `[[uuid:item_identifier]]`, where `item_identifier` should be replaced with the unique item identifier itself (for example, `item_1` or `icecream_question`).
- The coding scheme identifier for cognitive interview codes, which is `ci` and which is separated from the code identifier by `--`. For example, to code a problem with comprehension, the pattern `[[ci--comprehension]]` can be added.

Both unique item identifiers and cognitive interview codes can only contain lowercase and uppercase Latin letters (`a-z` and `A-Z`), Arabic numerals (`0-9`), and underscores (`_`). They cannot contain spaces or other special characters. If you do use those, the identifiers or codes containing them will be ignored.

This is an example of a very brief source that represents notes from a cognitive interview as coded using the ROCK standard:

```
###-----
```

```
How large is your family? [[uuid:familySize_7djdy62d]]
```

```
Participant also counts 3 dogs and 7 goldfish [[ci--understanding]]
```

```
Participant does not count own brothers and sisters, only their partner and children [[ci--retrieval]]
```

```
###-----
```

```
How many windows are there in your house? [[uuid:windows_7djdy62d]]
```

```
Participant does not live in a house, but in an apartment [[ci--content_adequacy]]
```

Participant also counts windows in doors inside the house `[[ci--understanding]]`

###-----

To specify codes in a source, you can use iROCK (<https://i.rock.science>; see The iROCK interface, chapter 14 of this version of the book, for an introduction) if you prefer to drag and drop codes, or alternatively, you can use whichever plain text editor you prefer (or you can even use a word processor such as Microsoft Word or LibreOffice Writer, as long as you remember to save the source in plain text format).

Once you have coded one or more sources, you can analyse these using software that implements the ROCK standard, such as a Shiny ROCK app or the `rock` R package.

## 18.1 Analysing cognitive interviews using the Shiny ROCK Beryl

A simple Shiny ROCK app exists to produce a heatmap from a single coded cognitive interview transcript or set of notes. You can access it at <https://preciousrock.shinyapps.io/shiny-rock-beryl>.

The Beryl app only processes the five codes from the Peterson, Peterson & Powell (2017) coding scheme: `understanding`, `retrieval`, `judgment`, `response`, and `content_adequacy` (see Cognitive interviews, chapter 4 in this version of the book). In addition, Beryl requires you to use unique item identifiers (uuids) to mark which part of the source covers which item. This is an example of a valid source that Beryl can process:

As you see, each item text is marked with a uuid (specifically, `[[uuid:familySize_7djdy62d]]` and `[[uuid:windows_7djdy62d]]`), and each cognitive interview coder is marked as such, too (specifically, `[[ci--understanding]]`, `[[ci--retrieval]]`, and `[[ci--content_adequacy]]`).

To use the app, open the second tab (marked “Heatmap”) and copy-paste the text of your coded source (or the example source shown above) into the large text box. Beryl will then show the heatmap below.

You can then tweak the dimensions of the heatmap (width, height, and font size) until you are happy, and download the result. For a high-quality file for sharing, you may want to download the PDF version; if you still want to make manual edits using Inkscape, you may want to download the SVG version; and if you want to embed the figure into a word processor document (e.g. LibreOffice Writer or Microsoft Word), you may want to download the PNG version.

## 18.2 Analysing cognitive interviews using the Shiny ROCK Amethyst

A slightly more advanced Shiny ROCK app for analysing cognitive interviews is called Amethyst. You can access it at <https://preciousrock.shinyapps.io/shiny-rock-amethyst>.

Once the app loaded, first upload your sources. To do this, click the “Sources” tab at the top and on that page, click the “Browse...” button to select one or more `.rock` or `.txt` files, or one `.zip` file.

Once the sources are uploaded, you can parse them to process all codes. To do this, click the “Parse Sources” button. If the sources can be parsed successfully, two additional tabs will appear at the top: “Qualitative Data Table” and “Cognitive Interviews”. You can ignore the first; you don’t need it for processing cognitive interviewing results. Therefore, click on the last tab, “Cognitive Interviews”.

There, you can produce the Cognitive Interview Heatmap. To do that, simply press the “Generate heatmap” button. The heatmap will appear, and you can download it using the “Download heatmap” button. This will download a `.png` file to your hard disk.

This heatmap shows your unique item identifiers in its rows and the codes you applied in the columns. This provides a quick overview of the types of problems exhibited by each item, quickly allowing you to spot which items are the most and the least problematic.





## Chapter 19

# Using the ROCK for a Qualitative Network Approach

The ROCK standard and the R `rock` package support Qualitative Network Approaches (QNA). This is an approach to coding qualitative data where instead of using hierarchical code structures, networked code structures are used. This can be useful, for example, when studying causal or structural associations expressed in the data.

Network codes have the form `[[from->to||type||weight]]`, where the last part is optional (i.e. `[[from->to||type]]` is also valid). The `from` and `to` parts contain the code identifiers that end up as nodes in the network. The `type` part indicates the kind of relationship between these two nodes, and the optional `weight` part indicates the weight of that relationship.

For example, the following

```
Coding qualitative data helps you understand the data better. [[coding->understanding||causal]]
And if you understand the data better, you can report your findings better. [[understanding->better_r||causal]]
However, coding data is also quite a risky business, for two reasons. [[coding->risks||causal]]
First, coding qualitative data can be quite tiring. [[coding->tiring||causal]] [[tiring->risks||structural]]
But, more risky, second, often it's quite the opposite: you feel energized after a coding bout. [[coding->energized||causal]]
And then, if you're not careful, that will keep you coding and coding and coding. [[carelessness->endless_coding||causal]]
That can be dangerous as you can forget to eat, drink, or sleep. [[endless_coding->self_negligence||causal]]
```

```
## PhantomJS not found. You can install it with webshot::install_phantomjs(). If it is installed, please
```

The default settings collapses

By default, multiple edges are collapsed

### 19.1 Customizing network appearance

You can specify how you want the network to look in a YAML fragment. The structure of the YAML fragment has to be as follows:

```
---
ROCK_network:
  edges:
    -
      type: causal
      color: black
      style: solid
    -
      type: causal_pos
      color: green
    -
      type: causal_neg
      color: red
    -
      type: structural
      color: black
      style: dashed
---
```

Every element describing a type of edge must contain the specification of the type it describes and a list of valid GraphViz edge attributes (see <https://graphviz.org/doc/info/attrs.html> for a list of GraphViz attributes).

## Chapter 20

# Using the ROCK for Decentralized Construct Taxonomies

### 20.1 The importance of clear coding instructions

When studying humans, one must deal with the somewhat challenging fact of life that one often does not study natural kinds. The objects of study are generally variables that are assumed to exist in people's psychology, usually called constructs. Those constructs are not assumed to exist as more or less modular, discrete entities (Peters & Crutzen, 2017). Instead, these constructs concern definitions that enable consistent measurement and consistent manipulation of certain aspects of the human psychology, without the pretense that the constructs are somehow clearly distinguished from other constructs.

As a consequence, data collection and analysis in research with humans differs fundamentally from data collection in sciences that do deal with natural kinds. Specifically regarding qualitative data, this lack of natural kinds further complicates the challenges that come with having humans code rich, messy data. Human perception and processing are flawed enough as it is: humans are, fortunately, no robots or artificial intelligences. The same capability for reading meaning and context in qualitative data that makes humans indispensable in coding also comes with risks. Without the existence of discrete, modular, objectively existing entities to code, the coding instructions become the only tangible foothold coders can rely on.

Therefore, being able to engage in the scientific endeavour with any degree of consistency over studies requires unequivocal communication about the constructs under study (Eronen & Bringmann, 2021). However, many theories do not provide sufficiently explicit definitions of the described constructs. Instead, there is often much room for interpretation: room that manifests as heterogeneity in constructs' definitions, operationalizations, and instructions for coding the constructs.

In the broader sense, it has been argued that this heterogeneity is a feature, not a bug (Devezer et al., 2019; Muthukrishna & Henrich, 2016; Ogden, 2016; Zollman, 2010). Heterogeneity in construct definitions and instructions for coding is not problematic, and to a degree, it is inevitable for different people to have different definitions of constructs, and therefore to code data differently.

However, if a group of researchers collaborates on a study, or if different groups of researchers aim to contribute to a cumulative knowledge base about a topic, such heterogeneity does become problematic if it remains unknown. The problem is illusory agreement about what exactly is being studied. If researchers fail to explicitly discuss their definitions and the corresponding coding instructions, it is very easy to remain under the impression that everybody has the same definitions and coding parameters in mind. Such illusory agreement is problematic within a group of collaborators doing a study and prevents knowledge accumulation of multiple studies.

For a lot of qualitative research, therefore, having comprehensive coding instructions that accompany comprehensive definitions of the constructs being coded (and often, corresponding procedures for obtaining qualitative data relating to those constructs) is one of the most important parts of qualitative research. Without the ability to unequivocally refer to specific construct definitions and the corresponding coding instructions, there is no guarantee that all involved researchers are coding the same constructs, even if everybody uses the same names.

## 20.2 Introduction to Decentralized Construct Taxonomies

To facilitate unequivocal references to specific definitions of constructs, combined with coherent instructions for operationalisation and coding, Decentralized Construct Taxonomy specifications (DCTs) were developed. DCTs are simple plain text files in the YAML format that specify, for one or more constructs:

- A unique identifier for the construct, the Unique Construct Identifier (UCID);
- A human-readable label (title / name) for the construct (which doesn't need to be unique, as the identifier is already unique);
- An exact definition of the construct;
- Instructions for developing a measurement instrument to measure the construct;
- Instructions for coding measurement instruments as measurement instruments that measure this construct;
- Instructions for generating qualitative data pertaining to this construct;
- Instructions for identifying when qualitative data pertains to this construct and then coding it as such.

### 20.2.1 Consistency over studies

DCT specifications can easily be re-used in different studies, for example in all studies in the same lab, in the same faculty, or organisation. To this end, a decentralized repository has been created. One instance that contains a number of DCT specifications for psychological constructs is available at <https://psycore.one> (this stands for **P**sycho**l**ogical **C**onstruct **R**epository). You can look at the coding instructions for the constructs available in that repository at <https://psycore.one/coding-qualitative-data>. Each of the DCTs in this repository has a unique URL with the construct's definition and corresponding instructions, for example [https://psycore.one/construct/?ucid=expAttitude\\_evaluation\\_73dnt5z2](https://psycore.one/construct/?ucid=expAttitude_evaluation_73dnt5z2).

Any study can re-use these DCTs by listing their Unique Construct Identifiers (UCIDs; for this last example, `expAttitude_evaluation_73dnt5z2`) and including the associated specifications with articles about the study (see below for the explanation of the file format). For constructs that have already been published in a repository, the unique URLs can also be included to provide a more userfriendly interface (note that since the repositories may not be persistent, it remains important to include the files with the DCT specifications).

## 20.3 Creating a DCT

### 20.3.1 Thinking about constructs

Creating a DCT requires knowing which construct you want to describe and what exactly the construct is and is not. This seems trivial - most researchers working with constructs (e.g., psychologists) rely on the assumption that they have sufficient tacit knowledge of the constructs they work with. However, because this knowledge never has to be made explicit, this assumption is never tested. Producing a

DCT for a construct confronts one with exactly how much one knows about a construct. Based on our experience, this is usually depressingly little.

The reason for this is that theories and the textbooks describing them usually do not provide clear definitions, either. In fact, that is one of the causes of the heterogeneity that exists. To a degree this is inevitable because constructs are not directly observable, and often do not represent natural kinds. But to a degree it can be remedied - by being very explicit about a construct's definition, by producing a DCT. Thus, while producing a DCT may not necessarily be easy, it is definitely worthwhile.

When creating DCTs, it is important to keep in mind that there are no objectively wrong or right "answers". After all, the constructs do not correspond to natural kinds. Various definitions can co-exist without any of them being wrong or right. In fact, since the constructs do not correspond to more or less discrete or modular entities anyway, one could argue that they are all 'wrong' (or are all 'right'). Given that at present, most constructs lack clear, explicit definitions, any explicitation is progress. And DCTs can always be updated or adjusted by updating their UCID. If you end up iterating through several versions, that's clear evidence that there was room for improvement in your original, implicit, definitions.

When creating a DCT, it doesn't matter where you start. If you have a pretty clear idea about the construct's definition, you start by making that explicit. But it's possible that while there are a number of measurement instruments for the construct (e.g. questionnaires), there is no clear definition available. In that case, you can start with the measurement instruments, too, and first complete the instruction for developing measurement instruments by deriving common principles from the measurement instruments you have.

In any case the process will be iterative. Eventually, you will complete at the definition of the construct, and probably at least two of the instructions (either the instruction for developing measurement instruments and for coding measurement instruments; or for developing manipulations and for coding manipulations; or for eliciting ('developing') qualitative data and for coding qualitative data). As you complete these sections, you will probably need to update other sections to make sure everything stays coherent.

On the surface, producing a DCT just consists of putting stuff in words. After all, you just need to type in the construct's name, definition, and add the instructions that allow you (and others) to work with the construct. This can be done within an hour. Most time is not spent on specifying the DCT in a file, but on arriving at definitions and instructions that you and your colleagues agree on. However, that is time well-spent.

By discussing the constructs you work with and the varying definitions that everybody uses, you achieve consensus. If you don't manage to achieve consensus about a given construct, that's fine of course - simply create two DCTs for two different constructs. You can even give them the same name - as long as they have different identifiers (UCIDs). If after these discussions, all researchers and their supervised students within your lab use the DCTs you produced, all research will be consistent. Of course, researchers without DCTs will often assume such consistency as well. And if they are right, the process of producing DCTs should be effortless. If the process proves more cumbersome, clearly it was necessary.

### 20.3.2 Description of edge cases

Clear definitions are most valuable when edge cases are encountered. For example, most people will have little difficulty in identifying 'chairs' and agreeing whether an object is a chair even without first explicitly communicating about and calibrating the definitions they use. It is with edge cases such as seating furniture with one, two, or three legs, or furniture that seats two or three people, where unclear definitions become problematic.

For example, a definition of a chair could be "A piece of furniture designed to support a sitting human". In this case, a bicycle would fall under this definition, and in a qualitative study, would therefore be

coded as a `[[chair]]`. This example is easily solved by updating the definition to “A piece of static furniture designed to support a sitting human”. However, in this definition, a bar stool with one leg would also be coded as `[[chair]]`, which in this case might fall beyond the intended definition. Describing all specific edge cases explicitly in the definition may make the definition unwieldy.

Therefore, the specific instructions in a DCT normally discuss edge cases explicitly, referring the user to alternative codes where appropriate. For example, the coding instructions for coding a piece of qualitative data as `[[chair]]` could include the sentence “Note that furniture without back and arm support and having three legs or less should not be coded as `[[chair]]` but instead as `[[stool]]`”.

Thus, coding instructions are often most useful if they do not only describe the core of a construct, but if they pay special attention to the periphery of a construct’s definition. Coding errors often concern ambiguity, and coding instructions should not add to this ambiguity.

### 20.3.3 Creating a DCT file

To create a DCT file, you can use any software that can create plain text files, such as Notepad, Textedit, Notepad++, BBEdit, Vim, Nano, or the RStudio IDE. A DCT file contains one or more DCT specifications, delimited by a line containing only three dashes (“---”). This is an example of an extremely simple DCT specification:

```
---
dct:
  dctVersion: 0.1.0
  version: 1
  id: chair_75v1264q
  label: "Chair"
  definition:
    definition: "A piece of furniture designed to support a sitting human."
  measure_dev:
    instruction: ""
  measure_code:
    instruction: ""
  aspect_dev:
    instruction: ""
  aspect_code:
    instruction: "Objects that have legs and a surface that was designed for humans to sit on. Note t
rel:
  id: furniture_75v125k8
  type: "semantic_type_of"
---
```

This example only specifies the UCID, name (label), definition, and instructions for coding, as well as one relationship to another construct with UCID “`furniture_75v125k8`” that this construct is apparently a type of. These relationships are parsed when the `rock` package reads a set of DCT specifications, and they are used to build a hierarchical tree of constructs (i.e. a deductive coding structure). You could omit these relationships of course, if you will not need to collapse codes or fragments based on higher levels in the hierarchy.

Instad of writing the DCT file by hand, you can also use the R package `psyverse`, specifically the functions `psyverse::dct_object()` and `psyverse::save_to_yaml()`, to create and save a DCT specification. For example, the above DCT specification can also be created using these two commands (after you installed the `psyverse` package):

```

myDCTspec <-
  psyverse::dct_object(
    prefix = "chair",
    label = "Chair",
    definition = "A piece of furniture designed to support a sitting human.",
    aspect_code = "Objects that have legs and a surface that was designed for humans to sit on. Note",
    rel = list(list(id = "furniture_75v125k8",
                    type = "semantic_type_of"))
  );
myDCTspec_asYAML <-
  psyverse::save_to_yaml(
    myDCTspec,
    file = tempfile()
  );

```

## 20.4 Coding with DCTs

When coding with DCTs, you code slightly differently than when you code without DCTs. Regular codes are simply delimited by two square brackets, e.g. `[[chair]]`. However, if you use DCTs, you specify this in the code: `[[dct:chair_75v1264q]]`. You can still combine this with inductive coding, for example for indicating that an important subtype of chairs are the thrones: `[[dct:chair_75v1264q>throne]]`. Like normal inductive codes, you can keep on nesting such subcodes infinitely to indicate ever more precise subconstructs, if need be (although one level will usually suffice).

## 20.5 Analysing DCT-coded sources





## Part IV

# Appendices and references



# Chapter 21

## References

- Babbie, E. (2007). *The Practice of Social Research*. Wadsworth.
- Borsboom, D., Cramer, A. O. J., Kievit, R. A., Scholten, A. Z., & Franić, S. (2009). The end of construct validity. In *The concept of validity: Revisions, new directions, and applications* (pp. 135–170). IAP Information Age Publishing.
- Borsboom, D., Mellenbergh, G. J., & Heerden, J. van. (2004). The Concept of Validity. *Psychological Review*, 111(4), 1061–1071. <https://doi.org/10.1037/0033-295X.111.4.1061>
- Bressler, D., Bodzin, A., Eagan, B., & Tabatabai, S. (2019). Using epistemic network analysis to examine discourse and scientific practice during a collaborative game. *Journal of Science Education and Technology*, 28(5), 553–566.
- Denzin, N., & Lincoln, Y. (2000). *Handbook of Qualitative Research*. Sage Publications.
- Devezer, B., Nardin, L. G., Baumgaertner, B., & Buzbas, E. O. (2019). Scientific discovery in a model-centric framework: Reproducibility, innovation, and epistemic diversity. *PLOS ONE*, 14(5), e0216125. <https://doi.org/gf86cs>
- Eronen, M. I., & Bringmann, L. F. (2021). The Theory Crisis in Psychology: How to Move Forward. *Perspectives on Psychological Science*, 1745691620970586. <https://doi.org/ghw2x3>
- Gelman, A., & Loken, E. (2014). The garden of forking paths: Why multiple comparisons can be a problem, even when there is no “fishing expedition” or “p-hacking” and the research hypothesis was posited ahead of time. *Psychological Bulletin*, 140(5), 1272–1280. <https://doi.org/dx.doi.org/10.1037/a0037714>
- Kane, M. (2013). The Argument-Based Approach to Validation. *School Psychology Review*, 42(4), 448–457. <https://doi.org/10.1080/02796015.2013.12087465>
- Maul, A. (2017). Rethinking Traditional Methods of Survey Validation. *Measurement: Interdisciplinary Research and Perspectives*, 15(2), 51–69. <https://doi.org/10.1080/15366367.2017.1348108>
- Muthukrishna, M., & Henrich, J. (2016). Innovation in the collective brain. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 371(1690), 20150192. <https://doi.org/gfzkmd>
- Ogden, J. (2016). Celebrating variability and a call to limit systematisation: The example of the Behaviour Change Technique Taxonomy and the Behaviour Change Wheel. *Health Psychology Review*, 10(3), 245–250. <https://doi.org/gh25r5>
- Peters, G.-J. Y., & Crutzen, R. (2017). Pragmatic nihilism: How a Theory of Nothing can help health psychology progress. *Health Psychology Review*, 11(2). <https://doi.org/10.1080/17437199.2017.1284015>
- Peterson, C. H., Peterson, N. A., & Powell, K. G. (2017). Cognitive Interviewing for Item Development: Validity Evidence Based on Content and Response Processes. *Measurement and Evaluation in Counseling and Development*, 50(4), 217–223. <https://doi.org/10.1080/07481756.2017.1339564>
- Ruis, A. R., Rosser, A. A., Quandt-Walle, C., Nathwani, J. N., Shaffer, D. W., & Pugh, C. M. (2018). The hands and head of a surgeon: Modeling operative competency with multimodal epistemic network analysis. *American Journal of Surgery*, 216(5), 835–840.
- Shaffer, D. (2017). *Quantitative Ethnography*.

- Simmons, J. P., Nelson, L. D., & Simonsohn, U. (2011). False-positive psychology: Undisclosed flexibility in data collection and analysis allows presenting anything as significant. *Psychological Science*, 22(11), 1359–1366. <https://doi.org/10.1177/0956797611417632>
- Smith, J. A., & Osborn, M. (2008). Interpretative phenomenological analysis. In J. A. Smith (Ed.), *Qualitative psychology: A practical guide to research methods* (pp. 53–80). Sage Publications.
- Tourangeau, R., Rips, L. J., & Rasinski, K. (2000). *The Psychology of Survey Response*. Cambridge University Press. <https://doi.org/10.1017/CBO9780511819322>
- Willis, G. B. (1999). Cognitive Interviewing. A "how to" guide. *Evaluation*, 1–37. <https://doi.org/10.1525/jer.2006.1.1.9>
- Woolley, M. E., Bowen, G. L., & Bowen, N. K. (2006). The Development and Evaluation of Procedures to Assess Child Self-Report Item Validity Educational and Psychological Measurement. *Educational and Psychological Measurement*, 66(4), 687–700. <https://doi.org/10.1177/0013164405282467>
- Zollman, K. J. S. (2010). The Epistemic Benefit of Transient Diversity. *Erkenntnis*, 72(1), 17–35. <https://doi.org/dnjk7f>
- Zörgő, S., Peters, G., Porter, C., Moraes, M., Donegan, S., & Eagan, B. (2022). Methodology in the Mirror: A Living, Systematic Review of Works in Quantitative Ethnography. In *Advances in Quantitative Ethnography*. (Vol. 1522, pp. 144–159). Springer, Nature.