smobsc Documentation

Release 0.0.1

everyone

Aug 14, 2019

Contents

Editors Sage Boettcher | Dejan Draschkow | Jona Sassenhagen | Martin Schultze

Maintainers Aylin Kallmayer | Leah Kumle | Leila Zacharias

Citation Boettcher, S.E.P., Draschkow, D., Sassenhagen, J., & Schultze, M. (Eds.). *Scientific Methods for Open Behavioral, Social and Cognitive Sciences*. https://doi.org/10.17605/OSF.IO/X4BCZ

The movement towards open and reproducible science prides itself on its grassroots and bottom-up development. However, even in these transformative times, most junior scientists will not discover the value and functional nature of transparent scientific practice before their graduate studies. This discovery is accompanied by transition costs, bad practices, and wasted time. Our project aims at placing open and reproducible scientific methods at the heart of scientific education - the first undergraduate years. To accomplish this we will establish and edit a teaching book covering not only the theoretical foundations of scientific methods for social and cognitive sciences, but also the practical skills necessary for transparent practice. It will offer standard text book knowledge accompanied by, but also presented via challenging and entertaining practical sessions in the programming environments R and Python – both of which are open source and freely available. The dynamic nature of a GitHub-based website will allow this resource to be both searchable with autonomous sections, while still including a narrative - students will be able to complete all the necessary steps for preparing, conducting and presenting scientific work. We aim at providing a standalone fully-comprehensible online book with the ability to be integrated in different teaching formats: e.g., assisting lectures, theoretical seminars, practical workshops, student-organized tutorials, or self-paced online learning. We hope this resource will change the skillsets and thinking of developing researchers early on and see such a project as essential to achieving transparent, open and reproducible scientific disciplines.

CHAPTER 1

Chapters

1.1 Philosophical Foundations

Subchapters

1.1.1 What is probability?

Authors Timo Lorenz

Citation Lorenz, T. (2019). What is probability? In S.E.P. Boettcher, D. Draschkow, J. Sassenhagen & M. Schultze (Eds.). Scientific Methods for Open Behavioral, Social and Cognitive Sciences. https://doi.org/10.17605/OSF.IO/WT96Q

When we talk about probability, we are talking about a concept that is omnipresent in our life – from thinking about all the probabilities you have after finishing your work or the most basic of all probability examples, the 'mother of probability', gambling. The concept of probability is even more fundamental when we take a look at how we understand science and approach all parts of statistical methodology. Yet most of the time students cringe when the word probability is spoken.

To be fair, almost everyone´s ability to estimate probabilities in real life decision-making is very limited at best (**'Fhaner, 1977'_**) because we seem to be prone to deterministic instead of probabilistic causations (**'Frosch & Johnson-Laird, 2011'_**). This makes it even more important to spend some time with this concept. If we are that bad at estimating probabilities in our everyday life, we should at least get a better understanding of how we deal with it in science. In this chapter, we will have a look at where our idea of probability comes from, along with its two points of view – the so called 'objective' and 'subjective' probability – that are still debated in psychology (**'Benjamin et al. 2018'_**; **'Dienes, 2011'_**; **'Howard, Maxwell, & Fleming, 2000'_**; **'Taper & Lele, 2010'_**). We will try to grasp the concepts in this chapter, since much of classical statistical tests will be explained later – so we'll try to use as few formulas as possible. Before we have a look at the two opposing schools of probability, let´s have a look at the history and maybe some reasons why one school was dominant in psychology for a long time.

A brief history of gambling and egos

While we can find thoughts and solutions on probability from great names such as Galileo and other scientists, the notion that probability has a quantitative value between 0 and 1 - and with this forming our contemporary understanding of probability, emerged somewhere in the middle of the 17th century with the work of Pascal and Fermat (**'Galavotti, 2015'_**). The reason why these two came up with new ideas on probability was more mundane than you might think – it was gambling. According to **'Hacking (1975)'_** Pascal was asked by the Chevalier de Méré from the court of Louis XIV to solve 'problems about games of chance' (**'Hacking, 1975'_**, p. 57) – namely gambling – such as fair chances in dice tosses and how players should divide the stake when a game got interrupted. Pascal and Fermat had the idea that their concept of probability could not only be used to solve gambling problems but could also be used as a general model for reasoning under uncertainty. **'Hacking (1975)'_** puts it quite frankly into the words 'On the one side it [probability] is statistical, concerning itself with stochastic laws of chance processes. On the other side it is epistemological, dedicated to assessing reasonable degrees of belief in propositions quite devoid of statistical background' (p.12). Et voilà, we had the birth of our modern understanding of probability and with it the problem that we have a possible epistemic (belief or subjective) and an empirical (frequencies or objective) interpretation of it.

For some time having two sides of a coin was not that much of a problem and both possible interpretations coexisted. You will find all kinds of important work on probability from all the names you can find in a typical textbook on statistics – from Huygens to the Bernoullli family, Poisson, Gauss, Borel, Markov to Kolmogorov who in 1933 showed that the mathematical results of probability follow a basic set of axioms (**'Dienes, 2008'_**; **'Galavotti, 2015'_**). But a few names stand out in this history of probability – Bayes, Laplace, Fisher, Neyman and Pearson - because they are basic to our understanding of probability in statistics and its opposing school, so let´s zoom in on them.

Thomas Bayes (1702-1761) was a minister and a member of the Royal Society with only one publication under his name during his lifetime. He became the name giver of the 'Bayesian school' or the subjective probability when his friend Richard Price published a manuscript of his after his death. The manuscript was on the problem of how one may obtain the probability of a hypothesis given some data – something we would later call the Bayes' theorem. This concept was as simple as it was beautiful – We modify our opinion with objective information: Initial beliefs + recent objective data = new and improved belief (**'McGrayne, 2011'_**). We will have a closer look at this theorem later in this chapter.

Bayes' ideas were picked up by Pierre-Simon Laplace (1749-1827) in the early 19th century. Laplace was a probabilist and even referred to as the "Newton of France" for his work in physics and astronomy. In his lifetime he came up with two theories of probability that showed more or less the same results with very large samples (**'Galavotti, 2015'_**). Just on a quick side note – Laplace despised gambling.

Times changed; politics as well as personal animosities played their part as much as a shift in the idea what 'science' could do, and with it crashed the idea that there is a balance between subjective beliefs and objective frequencies. The new natural science wanted a 'route to certain knowledge', and how could this be subjective in any way? This led to scientists treating the two points of view in probability as diametrical (**'McGrayne, 2011'_**). Soon mathematicians should take over the field of probability, bringing with them the frequentist approach or objective probability and most terms, concepts and techniques we use in statistics today (**'Dienes, 2008'_**).

The most dominant name in this upcoming approach would become Sir Ronald Fisher (1890-1962), a geneticist, who coined the terms 'null hypothesis' and 'significance' (**'Dienes, 2008'_**). Though often deemed the originator of the tradition of using $p < .05$ as an arbitrary level to judge results as statistically significant, Fisher (1926, p. 85) was open in suggesting that each researcher should select a threshold in relation to the specific study. As a matter of fact, Gigerenzer et al. (**'Gigerenzer, 2004'_**; p. 392) suggest that "Fisher thought that using a routine 5% level of significance indicated lack of statistical thinking". Fisher, after working for years on randomization, sampling theory, experimental designs, tests of significance and analysis of variance, published a revolutionary manual for non-statisticians – Statistical Methods for Research Workers (**'Fisher, 1925'_**). This manual was a game changer and helped to turn the frequentist approach into the state of the art for years to come.

Aside from (or maybe because of) his smart mind, Fisher was in a constant ad-hominem feud with his colleagues, especially Karl Pearson (1857-1936) – a British mathematician and biostatistician and the name-giver of the Pearson correlation, his son Egon Pearson (1895-1980), a statistician, and his colleague Jerzy Neyman (1894-1981), a mathematician (**'McGrayne, 2011'_**). The last two developed the ideas of hypothesis testing and statistical inference

(**'Pearson & Neyman, 1928'_**; **'Neyman & Pearson 1933'_**). Even though Pearson and Neyman had a heated and ongoing feud with Fisher, the three of them still became the 'godfathers' of the so called *objective probability* approach and with it the still most ubiquitous method in the social and behavioral sciences - null hypothesis significance testing (NHST; **'Cumming et al., 2007'_**, **'Perezgonzalez, 2015'_**). Now that we have an idea on the who is who of probability, let´s go and have a look at the two schools of thought in detail. We will have a look at their logic and their basic concepts. Let us begin with the objective probability.

Objective probability

The so called objective probability or frequentism is still the most common logic in the social and behavioral sciences, from statistics courses to most journals in this field. There are two classical approaches to frequentism – Fisher significance testing and Neyman-Pearson hypothesis testing. Table 1 (taken from **'Kline, 2013'_**, p. 68) shows you the difference between the two, and that most frequentists today follow the Neyman-Pearson logic, which most people now know by the name null hypothesis testing (NHST). As you can see in the table, the Neyman-Pearson logic is in reality a hybrid of Fisher´s ideas and terms he coined and the hypothesis testing ideas of Neyman and Pearson (**'Dienes, 2008'_**). In this part, we will have a look at the logic and assumptions of these steps in NHST.

Fisher	Neyman-Pearson
1. State H_0 2. Specify test statistic 3. Collect data, calculate test statistic, determine p 4. Reject H_0 if p ist small otherwise, H_0 is retained	1. State H_0 and H_1 2. Specify α 3. Specify test statistic 4. Collect data, calculate test statistic, determine p 5. Reject H_0 in favor of H_1 if $p < \alpha$; otherwise H_0 is retained

Frequencies and infinity!

The objective probability approach follows the idea that the probability of something happening is not in the mind, or a subjective belief, but rather objectively exists in the real world and needs to be discovered (**'Dienes, 2008'_**). A subjective interpretation of probability is not compatible with science (**'Popper, 2013'_**).

The objective interpretation of probability is analyzed through a long-run relative frequency (**'von Mises, 1957'_**)– 'wherein probability is the relative frequency of a given attribute, that can be observed in the initial part of an indefinite sequence of repeatable events' (**'Galavotti, 2015'_**, p. 748). What does that mean? The classical example is a coin toss – imagine you want to know if a coin is fair, so chances are equal between heads and tails. You flip it 10 times and it comes up with an unequal number of times between heads and tails. Does that mean the coin is not fair? No, because it is possible that a coin could show heads 3 out of 10 times. You would need an infinite number of coin tosses to determine exactly, whether the coin is fair – any number of repetitions smaller than infinity will always be an approximation.

Pretty quickly you now realize that the idea of an infinite number of observations would be impossible, and that this is an idealization of what is actually possible. So how can we manage this problem, since most of us do not have the time for an infinite number of observations? Neyman-Pearson came up with an idea for that problem – setting up a set of decision rules for accepting and rejecting a hypothesis so that in the long run we will often not be wrong (**'Dienes, 2008'_**).

Let´s come up with a hypothesis (and a rejection rule)

This set of decision rules is quite strict (see Table 1 to refresh the rule set) and we will have a look at it from the beginning. The first thing we need to do is set up two hypotheses. The first one is the null hypothesis (H_0) and the

second one is the alternative hypothesis (H_1).

The H_0 are most commonly either nil hypotheses – where the value is 0 - or point hypotheses with a numerical value of a parameter. You would use the nil hypothesis when the parameter is unknown, for example in a new field of research. When you have an idea that the parameter will be some specific value other than zero, you would use the point hypothesis.

The H1 is a range hypothesis that can be either non-directional (two-tailed), predicting any result that is not included in the H0, or directional (one-tailed), predicting a value that is smaller or greater than the one included in H0. To give you an idea, here is an example: Given that

$H_0 = 0$, a non-directional H_1 would be $H_1 \neq 0$ and a directional H_1 would either be $H_1 < 0$ or $H_1 > 0$. All this is to be specified before the data is collected.

Now we need a line of rejection and this is α or the level of significance which is set at 0.05 (remember - the term significance and the convention of $\alpha = 0.05$ were Fisher's ideas) by most conventions in the social and behavioral sciences. In a given observation we can now calculate the p-value, and if this is below $\alpha = 0.05$, we would speak of statistical significance. Sometimes people confuse α and p (**'Hubbard, Bayarri, Berk, & Carlton, 2003'_**). **'Gigerenzer (1993)'_** helps us to differentiate these two by referring to p as the exact level of significance in the observation and α is the line that gives us the long-run probability error.

The basic idea here is that if the H_0 is true, and we would have an infinite number of observations, in the long run, we would falsely reject the H_0. I told you above, it is a system of decision that will help us to minimize errors in the long run. So let us have a look at the two possible types of errors here.

It is about the errors. . . long term errors

We have met α but there is another important possible long-term error – β. Remember, α is the long-term probability error that says when $\alpha = 0.05$ there is a 5% long-term error chance to reject the H_0 when it is true. This type of error is called the Type I error. In the binary thinking of accepting or rejecting a hypothesis, there must be a second possible error – accepting the null hypothesis when it is in false. This type of error is called Type II error or β. We can put this into a nice little 'formula': $\alpha = P(\text{rejecting } H_0 | H_0 \text{ is true})$ and $\beta = P(\text{accepting } H_0 | H_0 \text{ is false})$.

In the frequentist approach, you should control for both types of long-term errors and decide on an acceptable level for both. Since Fisher suggested the $\alpha = 0.05$ most people and journals tend to blindly follow this rule. **'Aguinis et al. (2010)'_** advise caution with that and to reflect on the desired relative seriousness of the Type I vs. Type II error, depending on your research. **'Neyman (1953)'_** suggested $\beta = 0.20$ as the highest possible value for β, and $\beta = \alpha$ as its lower floor. So how do we control for the Type II error? You need to

1. estimate the effect size that matches your statistical method - e.g. Cohens-d (**'Cohen, 1977'_**) when looking for mean differences – and which you think would be relevant in real life, given your theory is true and 2. do a prospective (a priori) power calculation.

Power is essentially $1 - \beta$. So if you decide you want to keep β at .05 you need a power of 0.95. There are free calculators and programs like G*Power (**'Faul, Erdfelder, Buchner, & Lang, 2009'_**; **'Faul, Erdfelder, Lang, & Buchner, 2007'_**) as well as good practical papers (e.g. **'Howell, 2012'_**; **'Murphy & Myors, 2014'_**) out there to help you calculate power. This will be discussed more in-depth in a later section of this volume. Interestingly, many researchers seem to spend a lot of thought on the Type I error but almost seem to ignore the possible Type II error (**'Brock, 2003'_**; **'Kline, 2013'_**; **'Sedlmeier & Gigerenzer, 1989'_**). Unfortunately, this is not the only problematic thing that occurs a lot – there are some serious misconceptions about the p-value as well.

Some more errors but mostly not planned ones.

The p-value is essential to most statistical tests in NHST. It is the probability (where the p in p-value comes from) of witnessing the observed result or even a more extreme value if the null hypothesis is true (see **'Hubbard and Lindsay, 2008'_**; **'Kline, 2013'_**). Unfortunately, many psychologists – from students to professors - often have

some misconceptions about the p-value (**'Badenes-Ribera, Frias-Navarro, Iotti, Bonilla-Campos, & Longobardi, 2016'_**; **'Badenes-Ribera, Frias-Navarro, Monterde-i-Bort, & Pascual-Soler, 2015'_**; **'Haller & Krauss, 2002'_**, **'Oakes, 1986'_**). This is so common that we should have a close look at this so that you will not make these mistakes in your career. **'Badenes-Ribera et al. (2016)'_** name the most common misconceptions: the 'inverse probability fallacy', the 'effect size fallacy', the 'clinical or practical significance fallacy', the 'replication fallcy' and **'Verdam, Ort, & Sprangers (2013)'_** expand this by adding the 'proof fallacy'.

The inverse probability fallacy is the belief that the p-value tells us the probability of the theory is true given the data - when really it is the other way around, and not at all interchangeable. Coming back to a basic formula - the fallacy here is to think that $P(\text{theory}|\text{data})$ while in truth it is $P(\text{data}|\text{theory})$ and one cannot infer the probability of one of these two just by knowing the inverse variant.

'Dienes (2011)'_ fills this theoretical approach with a rather bloody and graphic example which should make this seizable for you: The probability of being dead given that a shark has bitten off one's head - or $P(\text{dead}|\text{head bitten off by shark})$ - is 1. The probability of a head bitten off by a shark given one is dead - or $P(\text{head bitten off by a shark}|\text{dead})$ – is almost 0 since most people die of other causes. Therefore, one should not mistake $P(\text{data}|\text{theory})$ with $P(\text{theory}|\text{data})$.

The effect size fallacy is the false belief that the smaller the p-value, the larger is the effect (**'Gliner, Vaske, & Morgan, 2001'_**). Yet the effect size is not determined by the p-value but by its appropriate statistic and the confidence interval (**'Cumming 2012'_**; **'Kline, 2013'_**). Simply spoken, the p-value by itself gives you very little information about the effect size.

The clinical or practical significance fallacy is closely related to the effect size fallacy because it links a statistically significant effect with the idea that it is an important effect (**'Nickerson, 2000'_**). The truth is that a statistically significant effect can be without any clinical or practical importance. Just imagine two samples of one million people each are measured in height and the statistical test shows that they have a statistically significant difference in height. But in real life, they have a mean-difference of one millimeter – no one would say that a one millimeter height difference has any practical importance. **'Kirk (1996)'_** states that the clinical or practical importance of results should be described by an expert in the field, not presented by a p-value.

The replication fallacy is the false belief that the p-value gives you an exact idea about the replicability of the results. This fallacy even has people mistakenly thinking that the complement of p (i.e. $1 - p$) tells you the probability of finding statistically significant results in a replication study (**'Carver, 1978'_**). Unfortunately 'any p-value gives only very vague information about what is likely to happen on replication, and any single p-value could easily have been quite different, simply because of sampling variety' (**'Cumming, 2008'_**, p. 286).

The proof fallacy is the fallacy to think that when the null hypothesis is rejected, it proves that the alternative hypothesis is true because there can be possible alternative explanations. Furthermore, it is also a fallacy to think when the null hypothesis is not rejected, it proves that the alternative hypothesis is false because this just might be a consequence of statistical power (see **'Verdam et al., 2014'_**).

Conclusion

As you can see, the school of objective probability or frequentism is not without some serious pitfalls and yet it is still the most dominant framework used in the social and behavioral sciences. It has its own logic that unfortunately is so often misunderstood that some researchers go so far as to call for an abandonment of significance testing (e.g. **'Harlow, Mulaik, Steiger, 2016'_**; **'Kline, 2013'_**). Other authors (e.g. **'Cummings, 2013'_**) or the APA manual (**'APA, 2010'_**) demand the reporting of confidence intervals instead of or in addition to p-values. Strangely, this is what Neyman often did. He rarely used hypothesis testing in his own research but most of the time reported confidence limits around the estimates of his model parameters (**'Dienes, 2008'_**). **'Oakes (1986)'_** muses that some of the confusion in frequentism is due to fact that many researchers unknowingly have a subjective probability or Bayesian understanding of research. So it is time to see have a look at this approach and see if you are one of them.

Subjective probability

Introduction and the Bayes theorem

Remember that objective probability 'only' tells us something about inferences about long-run frequencies and their possible error rate but not about the probability of a hypothesis being right. But most people want to have some information on that as well. Just imagine you are leaving your apartment but before you do that, you look out the window and think 'What are the odds it might rain today?'. Would you grab an umbrella or not? You might base your decision on how you high you estimate the probability of rain to be on this day. Objective probability cannot help you in this case, because this is a single event, not a long-run frequency. The moment you make a decision thinking 'I think it may rain today, I'd better take an umbrella with me', you are in the realm of subjective probability.

Subjective probability is the degree of belief you have in a hypothesis (**'Dienes, 2008'_**). Of course it gets a little more complicated than that when we are talking about how to implement subjective probability into a statistical tool but the essence stays the same. The most basic notion here, before we get to the details, is that you have an inkling of the probability of a hypothesis. You might check some sources, collect some data – in our example, you might check the Weather Channel – but at the end of day, you have to decide if you think the probability of rain is high enough to take an umbrella with you.

Because most people are not really good at updating their personal beliefs in the light of new information (**'Sutherland, 1994'_**), we have to come up with a system that helps us to be more scientific. At this point we come back to Bayes and his friend Price who presented his work posthumously to the Royal Society. In this work, Bayes describes the fundamental logic to subjective probability – the Bayes' theorem (**'Bayes & Price, 1763'_**):

$$P(H|D) = P(D|H) \cdot \frac{P(H)}{P(D)}$$

Now, let us pick this apart:

- $P(H|D)$ is the posterior, the probability of a hypothesis given some

data - $P(D|H)$ is the likelihood or the probability of obtaining the data given your hypothesis - $P(H)$ is the prior, your belief about the hypothesis before you start collecting data - $P(D)$ is the evidence or the data

We will take a closer look at these components in a moment, but first some more general ideas: if you want to compare hypotheses given the same data, P(D) would be constant and you switch the formula above to:

$$P(H|D) \propto P(D|H) \cdot P(H)$$

Your posterior is proportional to the likelihood times the prior – and this is the basic tenet of Bayesian statistics. It simply tells you that you will update the prior probability of your hypothesis when you have some data and you will form a new conclusion – the posterior. In real human words this means – from a Bayesian point of view, your scientific inference is updating your beliefs in a hypothesis when you have some new data (**'Dienes, 2008'_**). Before we get a more detailed look at some important concepts, let us make a short excursion into the philosophy of science and give these new concepts some time to settle in your mind. Our excursion should make it clearer why so many scientists had a hard time with subjective probability, even when most of us are using it intuitively.

A philosophical excursion to Popper & Hume

When you think about the logic of the Bayesian approach, it is pretty close to inductive thinking – the process to come up with rules from observations. Let us take the famous swan argument here as an example. You see one white swan; and another one; and another one; and so on, and you come to the inductive conclusion that all swans are white. You have no guarantee that this rule is true but due to your observations it seems plausible to you. You can do the same thought experiment with the thought that you will wake up the next morning or that the sun will rise. The school of thought that used inductive thinking was called positivism and this thinking had two famous opponents – David Hume (1711-1776) and Sir Karl Popper (1902-1994).

David Hume was a Scottish philosopher who argued that we should never reason from experience (seeing a lot of white swans) about situations we have not experienced yet (seeing a swan of a different color). You might say that in

your experience the probability increases when you see tons of white swans that the next one will be white too. Hume would disagree with that because it does not follow logically. Take the second thought experiment – you waking up in the morning. Every day you wake up in the morning and this experience should increase the probability of you waking up tomorrow – inductively speaking. Now, add age to the equation and you see at one point, it becomes less likely that you will wake up the next morning. Hume points out that 'no matter how often induction has worked in the past, there is no reason to think it will work ever again. Not unless you already assume induction, that is' (**'Dienes, 2008'_**, p.5). A historical fun fact that is closely related to the swan argument, comes from the time that the British went to Australia. Guess what they found? Of course, they found black swans.

The second interesting mind here is Karl Popper, who essentially started the research field of philosophy of science by attempting to formalize what distinguishes science from non-science. Popper argued against positivism and with it inductive thinking. His alternative philosophy was *fallibilism*. In a nutshell (because **'Chapter 1.2'_** is dealing with this in a much deeper way): You cannot verify a theory - say a theory is true; you can only falsify theory. In this, Popper agreed with Hume's pessimism on induction (e.g. **'Popper, 1934'_**). For him a theory would in a best case scenario always be a guess, nothing more.

As [Popper would later explain](https://www.youtube.com/watch?v=ZO2az5Eb3H0), a major contributor to his theory was a series of events that occured in his youth. On one hand, the young Popper admired how science had resolved the conflict between Newtonian Mechanics, and Einstein's Theory of Relativity. In 1919, a [daring experiment](https://en.wikipedia.org/wiki/Solar_eclipse_of_May_29,_1919) brought forceful support for Einstein's side. Einstein's theory had entailed that the gravitational pull of the sun should shift the visible appearance of remote stars around its disc - occluded by the eclipse of 1919 - would shift by a precise amount. The amount predicted by Einstein was slightly larger than that predicted by Newtonian physics. When the results arrived, Einstein's predictions were found to have withstood this test: Relativity was found to *not* conflict with empirical results. The test of Einstein's theory had not rejected, and thereby corroborate (as Popper would later call it), the theory. The similarities to null hypothesis testing are evident.

In contrast, Popper was disappointed with a growing dogmatism he found in his Marxist friends. They seemed disinterested in any contradictory evidence - only confirmation was what concerned them. Observing how much more succesful - epistemologically speaking - physics had been compared to Marxism, Popper suggested the latter model - and thus *falsificationism* - as the ideal according to which science should be judged: always aware of the impossibility of induction, always open, even embracing, the potential for falsification.

As you can imagine, these two, especially Karl Popper, had a huge influence on how people understood science in the 20th century, and why the school of objective probability was so dominant for a long time. Of course there is more to the story – from politics to history (for an in-depth look see **'McGrayne, 2011'_**) – but you have an idea why it took the school of subjective probability and with it the Bayesian approach for inductive reasoning so long to be back in the game. Now it is time for us to take a deeper look into the Bayesian ideas and its concepts.

The prior

Let us start at the beginning – the prior or for the formula aficionados - $P(H)$. Remember, the prior is your belief about the hypothesis before you start collecting data. How can we address this? First we have to assign a number between 0 and 1. Zero means there is no chance that the hypothesis is true and one means you are certain it is true. If you ask yourself how you should deal with all the possibilities between 0 and 1, the answer you will get from most people who have something to do with Bayes will be – How much money would you be willing to bet on your statement? This is a rather unclear answer so let us see how we can establish a prior in a more formal matter.

What we need is a distribution for the prior. First ask yourself if you have any previous information on the matter. This information may vary - from a special subjective belief to previous studies. If there is no information, we can use a 'uniform prior' or 'uninformed prior' with a uniform distribution where all values are equally likely. Do you have some previous information – let us say the distribution of the construct intelligence? You know that the distribution is a normal distribution with mean of 100 and a standard deviation of 15. So you could use this as your prior. Sometimes people use different priors to see how robust their posterior distribution is after the data. Some just use uninformed prior so that the likelihood (we will come to that one soon) will dominate completely – these researchers are called

'objective Bayesians' (**'Dienes, 2008'_**).

The concept of the prior is hard to grasp in the beginning and could be a big obstacle for some people to try Bayesian methods. And of course, there are a lot of debates about possible priors (e.g. **'Gelman, 2009'_**; **'Kruschke, 2010'_**; **'van de Schoot et al. 2014'_**; **'Vanpaemel, 2010'_**; **'Winkler, 1967'_**) because this is the most subjective part of this school of thought. If one person chooses a prior, it does not mean another person would agree with that prior. I hope you get the idea of the prior here.

Likelihood

Now that we know more about the prior $P(H)$, let us now talk about the second part – likelihood $P(D|H)$. The likelihood contains the information about the parameters given the data. This means that the support for our hypothesis is provided by our data by a likelihood distribution with the possible values (**'van de Schoot et al., 2014'_**). Remember the Bayes´ theorem from above? The posterior is proportional to the likelihood times the prior or $P(H|D) \propto P(D|H)P(H)$. The likelihood connects the prior to the posterior so all information that is relevant to inference from the data is provided by the likelihood (**'Birnbaum, 1962'_**). We will have a likelihood distribution that is combined with the prior distribution or $P(D|H)P(H)$ to obtain our posterior distribution $P(H|D)$. What does that exactly mean?

Go back to your idea of previous information on your question. If you had no information and you were using a non-informative prior with a uniform distribution, all results would be equally possible. If you combine this with the likelihood, then it will show you exactly the posterior distribution because every probability in the prior was the same. But if you have some information and you are using an informed prior with a distribution of your choice, the likelihood will be combined with that information to form a posterior distribution. In the second case it means that the hypothesis with the greatest support from the data – the greatest likelihood – might differ from the highest posterior probability distribution. Also, if you have a lot of data the influence of the prior becomes less important to the posterior distribution (*Dienes, 2008*). Let us have a look at this with an example.

Imagine you would be interested in the number of rainy days in January and you have no idea about rain (uninformed prior). You would collect data by looking out the window (data and likelihood), you would come up with an idea about how many days it would rain (posterior) and maybe use that knowledge next year in January as a new and slightly informed prior. Or in a second case, you have the belief that it rains mostly when it is grey and cloudy (informed prior). Most January days in Central Europe are grey and cloudy so according to your belief, it should rain a lot. Once again you are collecting data by looking out your window (data and likelihood) and let us assume, it does not rain much but it is grey and cloudy, and you must update your information. But still the informed prior that it should rain on days that are grey and cloudy has an influence on your posterior. If you had collected tons of data on grey and cloudy days, and at the same time there is little chance of rain, the data would provide much more information on your posterior, your new belief about rainy and cloudy days, than your prior, your initial belief. Once again you can imagine why the prior is so important (and debated, as mentioned above) because if the prior is misspecified, the posterior results are affected due to the compromise between likelihood and prior (**'van de Schoot et al, 2014'_**). Now that we have an idea of how prior and likelihood interact, we need to have a look at the last piece of the puzzle – the posterior $P(H|D)$. The posterior will be a distribution that is a combination of prior distribution and likelihood distribution and represents your updated belief. The posterior shows you an explicit distribution of the probability of each possible value (**'Kruschke, Aguinis, & Joo, 2012'_**). Now you could use your updated belief as a new prior and repeat the whole process to update your knowledge once more.

Conclusion Bayes

I guess this was a lot to think about so let us take a breath and revisit the concepts. Using Bayesian methods and therefore the subjective probability approach is a way to update your subjective beliefs by combining your belief about a hypothesis and the evidence, and all this with distributions or different probabilities of possible results. This is much more complex than a possible black and white answer where you reject or do not reject a hypothesis. But we have seen that the prior is a double edged sword. It helps us to use previous knowledge (and often we have knowledge on

things) but it can have an influence on our results because our previous knowledge might be very wrong and so we might choose a wrong prior. Given enough data this problem might not be so relevant but still it has been opening up debates in science for quite some time (e.g. **'Gelman, 2009'_**; **'Kruschke, 2010'_**; **'van de Schoot et al. 2014'_**; **'Vanpaemel, 2010'_**; **'Winkler, 1967'_**). Furthermore, the distributions of posterior probabilities might give a more complex picture of reality but often we are forced to make black and white decisions (decide if we want to pay for a medication or not) because we have to act. It is a different approach to probability and now you have heard of it as well. So let us end with some final thoughts.

Conclusion chapter

At this point I hope you have a better understanding of two points of view of probability that are common in the social and behavioral sciences. Of course there is much more to it; more formulas, more mathematics, and different statistical approaches but my goal was to give you a first idea of the concepts that are at the basis of so many different methods in statistics. Both points of view come with their own strengths, weaknesses and possible pitfalls. I do not want to argue for one or against the other but my hope is that you will understand that both points of view have a different aim, a different inference, and are sensitive to different things. You should be aware of your research question and the kind of probability that helps you to find an answer to this question. Do you need a black and white answer using objective probability or do you need a continuous distribution of posterior beliefs using subjective probability? Both probabilities come with a huge toolbox of applicable statistical methods (and some of them are discussed by my colleagues in this volume) and many of those methods can be used with both approaches. So chose your tool and scientific approach to each question you ask very careful and aware of the alternatives. I wish you a pleasant journey into the wonderful world of statistics.

References

Aguinis, H., Werner, S., Lanza Abbott, J., Angert, C., Park, J. H., & Kohlhausen, D. (2010). Customer-centric science: Reporting significant research results with rigor, relevance, and practical impact in mind. Organizational Research Methods, 13(3), 515-539.

American Psychological Association (2010). Publication Manual of the American Psychological Association (5th Edition). Wahsington, DC: American Psychological Association.

Badenes-Ribera, L., Frias-Navarro, D., Iotti, B., Bonilla-Campos, A., & Longobardi, C. (2016). Misconceptions of the p-value among Chilean and Italian academic psychologists. Frontiers in Psychology, 7, 1247.

Badenes-Ribera, L., Frías-Navarro, D., Monterde-i-Bort, H., & Pascual-Soler, M. (2015). Interpretation of the p value: A national survey study in academic psychologists from Spain. Psicothema, 27(3), 290-295.

Bayes, T. & Price, R. (1763). An essay towards solving a problem in the doctrine of chances. By the late Rev. Mr. Bayes, F.R.S. Communicated by Mr. Price, in a letter to John Canton, A.M.F.R.S. Philosophical Transactions, 53, 370-418.

Benjamin, D. J., Berger, J. O., Johannesson, M., Nosek, B. A., Wagenmakers, E. J., Berk, R., ... & Cesarini, D. (2018). Redefine statistical significance. Nature Human Behaviour, 2(1), 6.

Birnbaum, A. (1962). On the foundations of statistical inference. Journal of the American Statistical Association, 57(298), 269-306.

Brock, J. K. U. (2003). The 'power'of international business research. Journal of International Business Studies, 34(1), 90-99.

Carver, R. (1978). The case against statistical significance testing. Harvard Educational Review, 48(3), 378-399.

Cohen, J. (1977). Statistical power analysis for the behavioral sciences. Cambridge, MA: Academic Press

Cumming, G. (2008). Replication and p intervals: p values predict the future only vaguely, but confidence intervals do much better. Perspectives on Psychological Science, 3(4), 286-300.

Cumming, G. (2013). Understanding the new statistics: Effect sizes, confidence intervals, and meta-analysis. New York, NY: Routledge.

Cumming, G., Fidler, F., Leonard, M., Kalinowski, P., Christiansen, A., Kleinig, A., & Wilson, S. (2007). Statistical reform in psychology: Is anything changing?. Psychological Science, 18(3), 230-232.

Dienes, Z. (2008). Understanding psychology as a science: An introduction to scientific and statistical inference. New York, NY: Palgrave Macmillan.

Dienes, Z. (2011). Bayesian versus orthodox statistics: Which side are you on?. Perspectives on Psychological Science, 6(3), 274-290.

Galavotti, M. C. (2015). Probability theories and organization science: The nature and usefulness of different ways of treating uncertainty. Journal of Management, 41(2), 744-760.

Hacking, I. (1975). The emergence of probability: A philosophical study of early ideas about probability, induction and statistical inference. Cambridge, UK: Cambridge University Press.

Haller, H., & Krauss, S. (2002). Misinterpretations of significance: A problem students share with their teachers. Methods of Psychological Research, 7(1), 1-20.

Harlow, L. L., Mulaik, S. A., & Steiger, J. H. (2016). What if there were no significance tests?. New York, NY: Routledge.

Howard, G. S., Maxwell, S. E., & Fleming, K. J. (2000). The proof of the pudding: an illustration of the relative strengths of null hypothesis, meta-analysis, and Bayesian analysis. Psychological Methods, 5(3), 315.

Howell, D. C. (2012). Statistical methods for psychology. Belmont, CA: Cengage Learning.

Hubbard, R., Bayarri, M.J., Berk, K.N., & Carlton, M.A. (2003). Confusion over measures of evidence (p‘s) versus errros (α‘s) in classical statistical testing. American Statistician, 57, 171-178.

Hubbard, R., & Lindsay, R. M. (2008). Why P values are not a useful measure of evidence in statistical significance testing. Theory & Psychology, 18(1), 69-88.

Faul, F., Erdfelder, E., Lang, A.-G., & Buchner, A. (2007). G*Power 3: A flexible statistical power analysis program for the social, behavioral, and biomedical sciences. Behavior Research Methods, 39, 175-191.

Faul, F., Erdfelder, E., Buchner, A., & Lang, A.-G. (2009). Statistical power analyses using G*Power 3.1: Tests for correlation and regression analyses. Behavior Research Methods, 41, 1149-1160.

Fhaner, S. (1977). Subjective probability and everyday life. Scandinavian Journal of Psychology, 18(1), 81-84.

Fisher, R.A. (1925). Statistical Methods for Research Workers. London, UK: Oliver and Boyd.

Frosch, C. A., & Johnson-Laird, P. N. (2011). Is everyday causation deterministic or probabilistic?. Acta Psychologica, 137(3), 280-291.

Gelman, A. (2009). Bayes, Jeffreys, prior distributions and the philosophy of statistics. Statistical Science, 24(2), 176-178.

Gigerenzer, G. (1993). The superego, the ego, and the id in statistical reasoning. In G. Keren & C. Lewis (Eds.), A handbook for data analysis in the behavorial sciences: Vol. 1 Methodological issues (pp. 311-339). Hillsdale, NJ: Erlbaum.

Gigerenzer, G., Krauss, S., & Vitouch, O. (2004). The null ritual: What you always wanted to know about null hypothesis testing but were afraid to ask. In Handbook on Quantitative Methods in the Social Sciences. Sage, Thousand Oaks, CA.

Gliner, J. A., Vaske, J. J., & Morgan, G. A. (2001). Null hypothesis significance testing: effect size matters. Human Dimensions of Wildlife, 6(4), 291-301.

Kirk, R. E. (1996). Practical significance: A concept whose time has come. Educational and Psychological Measurement, 56(5), 746-759.

Kline, R. B. (2013). Beyond significance testing: Statistics reform in the behavioral sciences. Washington, DC: American Psychological Association.

Kruschke, J. K. (2010). What to believe: Bayesian methods for data analysis. Trends in Cognitive Sciences, 14(7), 293-300.

Kruschke, J. K., Aguinis, H., & Joo, H. (2012). The time has come: Bayesian methods for data analysis in the organizational sciences. Organizational Research Methods, 15(4), 722-752.

McGrayne, S. B. (2011). The theory that would not die: how Bayes' rule cracked the enigma code, hunted down Russian submarines, & emerged triumphant from two centuries of controversy. London, UK: Yale University Press.

Murphy, K. R., Myors, B., & Wolach, A. (2014). Statistical power analysis: A simple and general model for traditional and modern hypothesis tests. London, UK: Routledge.

Neyman, J. (1953). First Course in Probability and Statistics. New York, NY: Henry Holt.

Neyman, J., & Pearson, E. S. (1933). IX. On the problem of the most efficient tests of statistical hypotheses. Philosophical Transactions of the Royal Society of London. Series A, Containing Papers of a Mathematical or Physical Character, 231(694-706), 289-337.

Nickerson, R. S. (2000). Null hypothesis significance testing: a review of an old and continuing controversy. Psychological Methods, 5(2), 241.

Pearson, J., & Neyman, E. S. (1928). On the use, interpretation of certain test criteria for purposes of statistical inference: Part I. Biometrika. A, 20, 175-240.

Perezgonzalez, J. D. (2015). Fisher, Neyman-Pearson or NHST? A tutorial for teaching data testing. Frontiers in Psychology, 6, 223.

Popper, K.R. (1934). Logik der Forschung. Zur Erkenntnistheorie der modernen Naturwissenschaft. (Logic of scientific discvovery). Wien, AU: Springer.

Popper, K.R. (2013). Quantum theory and the schism in physics: From the postscript to the logic of scientific discovery. London, UK: Routledge.

Oakes, M. (1986). Statistical inference: A commentary fort he social and behavioural sciences. Chichester, UK: Wiley

Sedlmeier, P., & Gigerenzer, G. (1989). Do studies of statistical power have an effect on the power of studies?. Psychological Bulletin, 105(2), 309.

Sutherland, S. (1994). Irrationality: The enemy within. London, UK: Constable and Company.

Taper, M. L., & Lele, S. R. (Eds.). (2010). The nature of scientific evidence: statistical, philosophical, and empirical considerations. Chicago, IL: University of Chicago Press.

Van de Schoot, R., Kaplan, D., Denissen, J., Asendorpf, J. B., Neyer, F. J., & Van Aken, M. A. (2014). A gentle introduction to Bayesian analysis: Applications to developmental research. Child Development, 85(3), 842-860.

Vanpaemel, W. (2010). Prior sensitivity in theory testing: An apologia for the Bayes factor. Journal of Mathematical Psychology, 54(6), 491-498.

Verdam, M. G., Oort, F. J., & Sprangers, M. A. (2014). Significance, truth and proof of p values: reminders about common misconceptions regarding null hypothesis significance testing. Quality of Life Research, 23(1), 5-7.

Von Mises, R. (1957). Probability, statistics and truth. London, UK: George Allen & Unwin

Winkler, R. L. (1967). The assessment of prior distributions in Bayesian analysis. Journal of the American Statistical association, 62(319), 776-800.

1.2 Computer stuff

Subchapters

1.2.1 Computers

1.2.2 Setting up a scientific research environment

1.2.3 Version Control, Git and Github

1.2.4 Docker

1.2.5 Introduction to Programming with Python

Authors Peer Herholz

Citation Herholz, P. (2019). Introduction to Programming with Python. In S.E.P. Boettcher, D. Draschkow, J. Sassenhagen & M. Schultze (Eds.). Scientific Methods for Open Behavioral, Social and Cognitive Sciences. https://doi.org/10.17605/OSF.IO/V8NPH

This section is meant as a general introduction to Python and is by far not complete. The goal of this section is to give a short introduction to Python and help beginners to get familiar with this programming language. We recommend to follow this section in parallel with a more interactive tutorial by clicking on this button:

What is Python?

Python is a simple and powerful programming language.

- what it can be used for in a scientific way as well as industry applications (which companies use python etc.)

For an introduction to statistics in python take a look at this interactive tutorial:

For an introduction to visualization in python take a look at this interactive tutorial:

Basics

Indentation

Whitespace is important in Python. Actually, whitespace at the beginning of the line is important. This is called indentation. Leading whitespace (spaces and tabs) at the beginning of the logical line is used to determine the indentation level of the logical line, which in turn is used to determine the grouping of statements.

This means that statements which go together must have the same indentation. Each such set of statements is called a block. We will see examples of how blocks are important later on. One thing you should remember is that wrong indentation rises `IndentationError`.

Comments

Comments are anything behind a # symbol at the beginning of a new line. Python skips those parts of the program and comments therefor intended to help the reader of a program to understand what and why a specific statement in the code is doing. Keep in mind that this person can be yourself in 2 month so it is good practice to include comments explaining important decisions, details or problems.

Help and Descriptions

Using the function help we can get a description of almost all functions. Imagine we do not know what the `math.log()` function is doing or how to use it... we can just call the help function and find out.:

```
help(math.log)
```

Variables

Variable Names

Variable names in Python can contain alphanumerical characters a-z, A-Z, 0-9 and some special characters such as an underscore. Normal variable names must start with a letter. By convention, variable names start with a lower-case letter, and Class names start with a capital letter. Don't worry if you do not know what Classes are yet. We will get to that later.

In addition, there are a number of Python keywords that cannot be used as variable names. These keywords are:

and, as, assert, break, class, continue, def, del, elif, else, except, exec, finally, for, from, global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, while, with, yield.'

Note: Be aware of the keyword `lambda`, which could easily be a natural variable name in a scientific program. But being a keyword, it cannot be used as a variable name.

Assigning Variables

The assignment operator in Python is =. Python is a dynamically typed language, so we do not need to specify the type of a variable when we create one. Assigning a value to a new variable creates the variable.:

```
# assigning x to 1.0 and y to 3
x = 1.0
y = 3
```

Data Types

Although not explicitly specified, a variable does have a type associated with it. The type is derived form the value it was assigned and defines the operations that can be done with a specific variable.

You can check the data type with:

```
# check type of x
type(x)
```

Python has four standard data types that we will briefly go over now.

Numbers

This data type stores numerical data and python supports different numerical data types including:

```
# integer
x = 1

# float
x = 1.0
```

String

Strings are the variable type that is used for storing text messages. They are an array of characters and are indicated through either single or double quotation marks.

```
# string
s = "This is a string"
s2 = 'This also is a string'
```

You can specify multi-line strings using triple quotes - (`"""` or `'''`). You can use single quotes and double quotes freely within the triple quotes. An example is:

```
'''This is a multi-line string. This is the first line.
This is the second line.
"What's your name?," I asked.
He said "Bond, James Bond."
'''
```

Strings can be formatted and handled in multiple ways. For example, we can index a string using []:

```
# this gives us the first character in s ("T")
s[0]
```

Heads up MATLAB user: Indexing starts at 0!

We can also extract a part of a string using the syntax [start:stop], which extracts characters between index start and stop. This is called *slicing*:

```
# output of this will be "This"
s[0:4]
```

If we omit either (or both) of start or stop from [start:stop], the default is the beginning and the end of the string, respectively:

```
# This hands us "This"
s[:4]

# This hands us "is a string"
s[4:]
```

We can also define the step size using the syntax [start:end:step] (the default value for step is 1, as we saw above):

```
# entire string, step size of 1
s[::1]

# entire string, step size of 2. Every second character will be selected
s[::2]
```

Python has two string formatting styles. An example of the old style is below, specifier %.2f transforms the input number into a string, that corresponds to a floating point number with 2 decimal places and the specifier %d transforms the input number into a string, corresponding to a decimal number.

```
# s2 = 'value1 = 3.14. value2 = 1'
s2 = "value1 = %.2f. value2 = %d" % (3.1415, 1.5)
```

The same string can be written using the new style string formatting.

```
s3 = 'value1 = {:.2f}, value2 = {}'.format(3.1415, 1.5)
```

There are a lot more useful operations that can be done on strings and some of them can be explored in the interactive part of this introduction.

Lists

Lists are very similar to strings, except that each element can be of any type. A list entails items separated by commas and enclosed in square brackets `[]`.:

```
# list
l = [1, 2, 3]
l2 = ["one", "two", "three"]
```

We can use the same slicing techniques to manipulate lists as we could use on strings. Elements in a list do not all have to be of the same type and Python lists can be inhomogeneous and arbitrarily nested::

```
# also a list
l = [1, 'a', 1.0]

# nested list
nested_list = [1, [2, [3, [4, [5]]]]]
```

Lists play a very important role in Python, and are for example used in loops and other flow control structures (discussed below). There are number of convenient functions for generating lists of various types, for example the range function (note that in Python 3 range creates a generator, so you have to use list function to get a list).

```
start = 10
stop = 30
step = 2

list(range(start, stop, step))
```

Adding, inserting, modifying, and removing elements from lists

We can modify lists by assigning new values to elements in the list. In technical jargon, lists are mutable.

```
# assigning "p" to the second element in l
l[1] = "p"

# assigning "s" to the second and "m" to the third element of l
l[1:3] = ["s","m"]
```

Tupels

Tupels are very similar to lists. The main difference is that tupels are enclosed in parentheses and cannot be updated after they have been assigned.

```
# tupel
t = (1,2,3)
```

If we try to assign a new value to an element in a tuple we get an error.

```
point[0] = 20
```

```
TypeError:  'tuple' object does not support item assignment
```

Dictionaries

Dictionaries are also like lists, except that each element is a key-value pair. The syntax for dictionaries is `{key1 : value1, ...}`:

```
params = {"parameter1" : 1.0,
          "parameter2" : 2.0,
          "parameter3" : 3.0,}
```

Dictionary entries can only be accessed by their key name.

```
# accessing entry for key "parameter1"
params["parameter1"]

> 1.0

# changing an entry
params["parameter1"] = "A"

# adding a new entry
params["parameter4"] = "D"
```

Operators and Comparisons

Operators can change the value of operands. Python contains different types of operators and we will touch on two fundamental ones: Arithmetic and comparison operators.

Arithmetic Operators

Arithmetic operators include:

+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo
**	Power

Comparison Operators

These operators are used to compare their operands. They return either `True` or `False` depending if the condition under which the operands are compared applies or not.

==	evaluates if operands are equal
!=	evaluates if operands are not equal
>	evaluates if left operand is greater than the right operand
<	evaluates if right operand is greater than the left operand
>=	evaluates if left operand is greater or equal than the right operand
<=	evaluates if right operand is greater or equal than the left operand

Control Flow

Python usually executes code in an exact top-down order. However, that is not always what we want. Imagine a situation where different blocks of code should be executed depending on different situations, conditions or decision. What if a different sound should be played depending on if participant correctly answered in a trial? Python provides different control flow statements to achieve exactly that.

Conditional statements: if, elif, else

The Python syntax for conditional execution of code use the keywords `if, elif (else if), else`: The `if` statement is used to check if a specific condition is met. If this is the case, the block followed the if-statement (if-block) is executed. If not this block of code is skipped.

```
x = 1
# check if x equals 1 and print the answer
if x == 1:
  print("Yes, x equals 1")

> "Yes, x equals 1"
```

The if-statement can be accompanied by an `elif` and/or `else` statement: In this case, python first checks the first if-statement. If this evaluates to "True" python executes the following indented code block and skips the rest of the if-elif-else statement as one of them already evaluated to true. Otherwise, python evaluates every statement in this block until one evaluates to true or it reaches the else-statement which gets executed if non of the if- or elif-statements evaluates to true.

```
x = 5

if x < 5:
  print("X is smaller than 5")
elif x >5:
  print("X is bigger than 5")
else:
  print("X equals 5")

> "X equals 5"
```

For the first time, here we encountered the mentioned indentation. This means that we have to be careful to indent our code correctly, or else we will get syntax errors.

for - loop

The `for` loop iterates over the elements of the supplied list (or any other iterable object), and executes the containing block once for each element. Any kind of list can be used in the for loop. For example:

```
for x in [1,2,3]:
  print(x)

> 1
  2
  3

for x in range(-1,1):
  print(x)

> -1
  0
  1
```

Sometimes it is useful to have access to the indices of the values when iterating over a list. We can use the enumerate function for this:

```
for idx, x in enumerate(range(-1,1)):
  print(idx, x)

> 0 -1
  1 0
  2 1
```

while - loop

The `while` loop allows to repeatedly execute a block of code as long as a specified condition is met. Python checks the condition and if it evaluates to `True` executes the while-block. It then checks the condition again, if it is still `True` the block is executed again, else python continues to the next statement in the block.

```
# while loop will stop as soon as i = 5
i = 0

while i < 5:
  print(i)

  i = i + 1
print("done")

> 0
  1
  2
  3
  4
  done
```

continue, break, pass

To control the flow of a certain loop you can also use `break`, `continue` and `pass`. `break` can be used to break out of a loop and force python to stop the execution of a loop statement. The `continue` statement is used to tell Python to skip the rest of the statements in the current loop block and to continue to the next iteration of the loop (i.e. start executing the loop-block from the beginning). `pass` basically tells python to do nothing and to carry on with executing the script.

```
rangelist = list(range(10))

for number in rangelist:
    # Check if number is one of
    # the numbers in the tuple.
    if number in [4, 5, 7, 9]:
        # "Break" terminates a for without
        # executing the "else" clause.
        break
    else:
        # "Continue" starts the next iteration
        # of the loop. It's rather useless here,
        # as it's the last statement of the loop.
        print(number)
        continue
else:
    # The "else" clause is optional and is
    # executed only if the loop didn't "break".
```

(continues on next page)

(continued from previous page)

```
    pass # Do nothing

> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
  0
  1
  2
  3
```

Functions

Functions allow to reuse pieces of code by assigning names to them. By calling a function by their name it is possible to use it anywhere in your program without having to repeat the code "hidden" behind the name of the function. There are a lot of build-in functions in oython. However, it is also possible to create your own functions. A function in Python is defined using the keyword `def`, followed by a function name, a signature within parentheses `()`, and a colon `:`. The following code, with one additional level of indentation, is the function body.

```
def say_hello():
  # block belonging to the function
  print('hello world')

say_hello() # call the function

> 'hello world'
```

Classes

Classes are the key features of object-oriented programming. A class is a structure for representing an object and the operations that can be performed on the object. In Python a class can contain *attributes* (variables) and *methods* (functions). A class is defined almost like a function, but using the class keyword, and the class definition usually contains a number of class method definitions (a function in a class). Each class method should have an argument `self` as it first argument. This object is a self-reference, meaning it is referring to the current instance of the class.

Some class method names have special meaning, for example:

`__init__` : The name of the method that is invoked when the object is first created. `__str__` : A method that is invoked when a simple string representation of the class is needed, as for example when printed. There are many more, see http://docs.python.org/3.6/reference/datamodel.html#special-method-names

```
class Point:
  """
  Simple class for representing a point in a Cartesian coordinate system.
  """

  def __init__(self, x, y):
      """
      Create a new Point at x, y.
      """
      self.x = x
      self.y = y

  def translate(self, dx, dy):
      """
      Translate the point by dx and dy in the x and y direction.
```

(continues on next page)

(continued from previous page)

```
        """
        self.x += dx
        self.y += dy

    def __str__(self):
        return("Point at [%f, %f]" % (self.x, self.y))

    # creating a new instance of a class
    p1 = Point(0, 0)  # this will invoke the __init__ method in the Point class
    print(p1)         # this will invoke the __str__ method

    > Point at [0.000000, 0.000000]
```

To invoke a class method in the class instance `point2` at coordinates `x=1`, `y=1`:

```
point2 = Point(1, 1)
print(point2)

> Point at [1.000000, 1.000000]

point2.translate(2, 2)
print(point2)

> Point at [3.000000, 3.000000]
```

Accessing values of a class object directly:

```
point3 = Point(1, 4)
print(point3.y)

> 4
```

Modules <— BIS HIER

Most of the functionality in Python is provided by modules. To use a module in a Python program it first has to be imported. A module can be imported using the import statement. For example, to import the module math, which contains many standard mathematical functions, we can do:

```
import math
```

This includes the whole module and makes it available for use later in the program. For example, we can do:

```
import math

x = math.cos(2 * math.pi)

print(x)
```

Importing the whole module is often times unnecessary and can lead to longer loading time or increase the memory consumption. Alternative to the previous method, we can also chose to import only a few selected functions from a module by explicitly listing which ones we want to import:

```
from math import cos, pi

x = cos(2 * pi)
```

(continues on next page)

(continued from previous page)

```
print(x)
```

It is also possible to give an imported module or symbol your own access name with the as additional:

```
import numpy as np
from math import pi as number_pi

x = np.rad2deg(number_pi)

print(x)
```

Exceptions

In Python errors are managed with a special language construct called "Exceptions". Such exceptions arise when Python cannot cope with a situation in the code. Imagine you are calling a function but the function does not exist. Depending on the issue on hand, Python raises different exceptions. If this exception cannot be handled immediately, the script terminates and quits.

Examples are:

NameError	Raised when an identifier is not found
SyntaxError	Raised when there is an error in the syntax
IndentationError	Raised when the indentation is not specified correctly

Python also provides the opportunity to protect your code from errors and exception by placing code snippes into a *try-block*.

try: # normal code goes here
except: # code for error handling goes here this code is not executed unless the code above generated an error

For example:

```
try:
  raise Exception("description of the error")
except(Exception) as err:
  print ("Exception:", err)
```

How does this work? All the code that might raise exceptions goes after then try-statement. Everything that goes after the except-statement is only executed if an error arises and is supposed to handle the error.

Finally-statement

Another useful extension of this concept is the finally- statement. It can be used to specify a block of code that should be executed wether an exception is raised or not. In other words, code that goes after an finally-statement is always executed.

```
try:
  print("test")
  # generate an error: the variable test is not defined
  print(test)
except Exception as e:
```

(continues on next page)

(continued from previous page)

```
  print("Caught an exception:" + str(e))
finally:
  print("This block is executed after the try- and except-block.")
```

File I/O

This section should give you a basic knowledge about how to read and write CSV or TXT files.

1.3 Conducting Experiments

Subchapters

1.3.1 Variables in Psychology

1.3.2 Contrafactual Causality

1.3.3 Experimental Designs

e.g. 2*2 Design gambling task

1.3.4 Programming Experiments in Python with Psychopy

Authors Jona Sassenhagen

Citation Sassenhagen, J. (2019). Programming Experiments in Python with Psychopy. In S.E.P. Boettcher, D. Draschkow, J. Sassenhagen & M. Schultze (Eds.). Scientific Methods for Open Behavioral, Social and Cognitive Sciences. https://doi.org/10.17605/OSF.IO/

Story Time: Galton's Pendulum

In the 1880s, scientific progress had opened up a novel entertainment option for Her Majesty Queen Victoria's subjects: have their psychometric traits measured in Francis Galton's Science Galleries, at the South Kensington Museum. Visitors paid "3 pence" (presumably some sort of currency) and then partake in a series of encounters with novel and strange machinery. Thousands of participants volunteered, establishing one of the first large-sample psychometric databases in history. Amongst the things measured was how long it took the curious museum goer to press a button in response to a simple light stimulus.

Galton measured response times using an ingenious design called the Pendulum chronograph. Simplified, the position of a swinging pendulum at the time point of a key-press could be compared to a reference table. This Galton used as an estimate of response time, which allowed a state of the art resolution of 1/100th of a second. Never before had mental process been measured with higher precision.

Other experimenters around this time were stopping response latencies with manual stopwatches. Galton himself conducted human geographic surveys by walking around London, mentally rating the women he saw for their attractiveness, and covertly noting the ratings with a self made device: a paper cross, into which he, in his pocket, punched a hole indicating if the woman was of below average, average, or above average "beauty".

It is not advised to mention such practices in a contemporary presentation of scientific results. At the very least, the practices would be seen as grossly inaccurate, and for many, highly open to experimenter bias. In particular, they lack

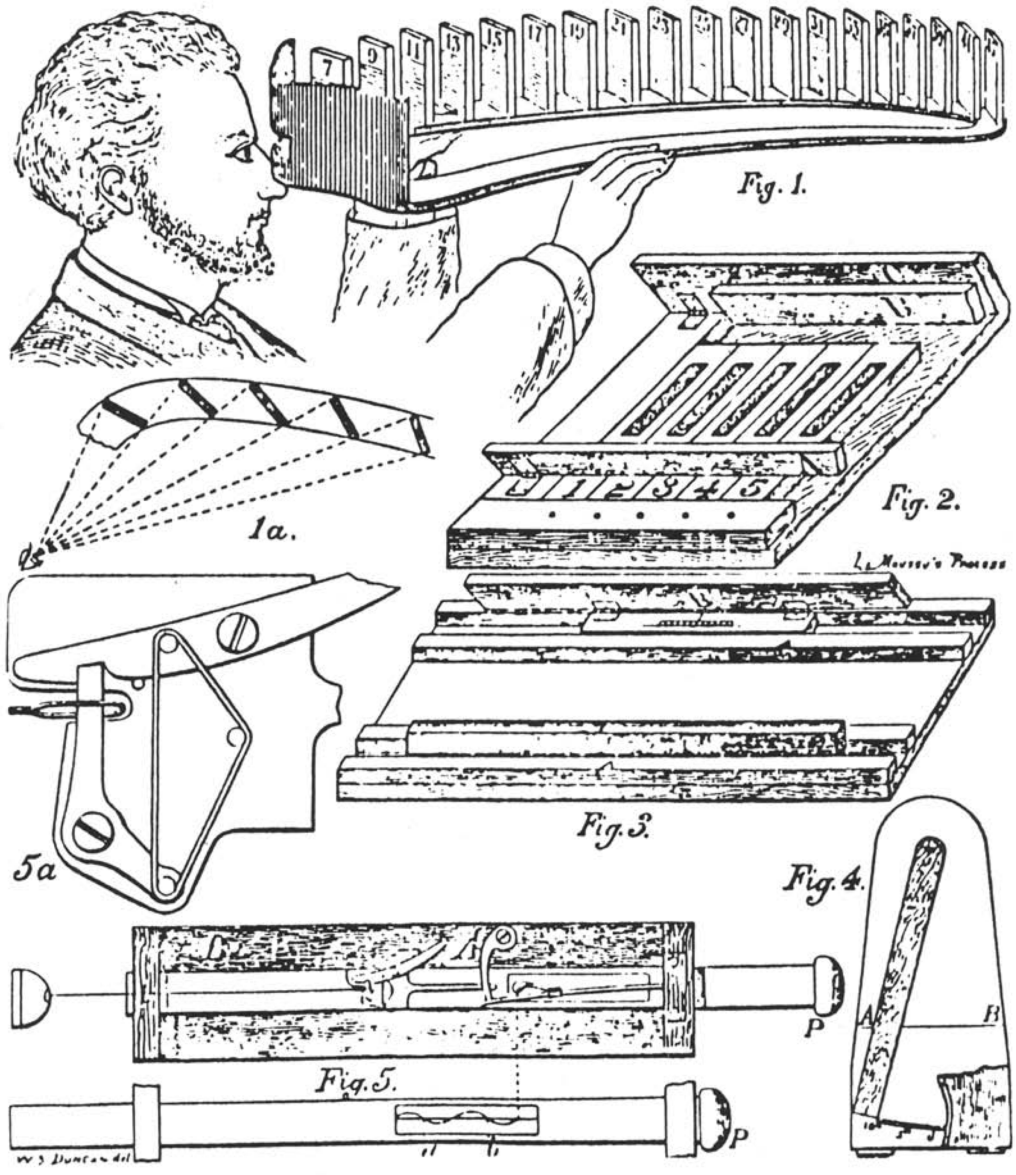

Fig. 1: Some of Galton's tools. The pendulum chronograph is more complicated than any of these!
Public domain image taken from (citation missing).

precision. Also, to be frank – they sound like a lot of work! While some might decry a lack of entrepreneurial spirit, most of us will readily admit the advantage of computerized methods.

Progress has made the work of the researcher much simpler. (Remember this sentence whenever you feel frustrated attempting to program an experiment! At least you are not Galton, keeping a pendulum chronograph in working condition, or walking across London to mentally file people into rather crude and subjective boxes.) Today, computers can show various visual input, play all kinds of sounds, and accurately measure ... well, a narrow kind of behavioral responses made by experimental participants. Our computer's graphics and sound cards, and the keyboard and mouse drivers, are relatively arcane. There is as of yet no convenient way to get a computer to show words or pictures on a screen for the purposes of psychological experiments from our favourite programming language, R. There are many commercial, closed-source solutions, which we will all ignore in favour of the powerful and open Python options.

Why? Perhaps the most important benefit computerized experiments have is that they are much more reproducible. Using an Open Source program and making experimental stimuli and scripts available online allows other researchers to 1. exactly retrace what happened in the original experiment, 2. repeat it ad libitum. To actually exploit this reproducibility potential, we must use software that is open. The biggest open source experimental presentation software is Psychopy [Pei07].

Programming Experiments in Python with Psychopy

Psychopy allows us to write simple and readable Python code to control our computer's low-level capacity for displaying and playing stimuli. Why is this necessary? Because we need to work with the computer on a low level in order to get it to achieve highly precise timings, and smoothly display even complex visual stimuli. That is one half of the experimental program; the other will consist in translating the (experimental design) into computer code, so that, e.g., a study participant is presented with the required number of trials resulting from your (power calculation) for the conditions resulting from your (latin square design).

Because Psychopy is written in Python, we having already learned Python, learning Psychopy reduces to learning the Psychopy-specific modules.

The basic logic of experiments in Psychopy

Stimuli

Psychopy can display auditory and visual stimuli; visual stimuli may be static, or dynamic (moving animations, or videos). To display visual stimuli, Psychopy must know about at least one `Monitor`, and at least one `Window`. Such a `Window` is the plane on which drawn stimuli will be shown. Note that `Monitor` and `Window` are software objects primarily inside of Psychopy. They allow Psychopy to show things in a window on your physical screen (or fullscreen).

Internally, Psychopy knows the backside and the front side of each *Window'*. When a `Window` is newly created, both sides will be empty. We can now draw things on the backside (using the `draw` method of various visual objects). Once everything we want to show has been drawn, the `Window` is "flipped" so that the painted backside is now shown on the physical screen. The new backside is blank. We can now draw other things on this blank backside. One option might be to draw nothing, to show a blank screen after the current stimulus. Eventually, we flip again; this clears everything we had drawn on the original backside, and shows the other side on the screen. So: we manually paint – piece by piece – one side of the window with everything we want to show, flip it to show it, paint the new backside, flip again to show and clear, and repeat.

— put image tbd by aylin here —

The visual stimuli we can paint on screens live in the `psychopy.visual` submodule. This includes various geometric shapes, as well as the `TextStim` and `ImageStim` classes, which we will discuss extensively in the following.

For movie stimuli, see the MovieStim class. Other stimuli include random dot motion and grating stimuli.

Keeping track of time and responses

Psychopy allows collecting button or keyboard responses and mouse events. For time tracking, one can create and use one or more `clock` objects. To time the duration of an event or interval, the clock is reset before the event/at the start of the interval, and then measured at the end. For example, to measure a response time to a stimulus, the clock is reset exactly when the window flip to show the stimulus happens. Then, the clock is checked when the button press happens. No pendulums involved!

Keyboard responses are measured by the `event.waitKeys` and `event.getKeys` functions. If provided with a clock, they return a Python `list` of `tuples`, each `(key, time_since_clock_reset)`

Storing results and experimental logic

Psychopy provides expensive functionality for logging results and the logistics of presenting stimuli, but it is not even required to learn these; basic Python code can be sufficient. For example, the humble `print` option can be employed to write strings (such as response events) to disc.

A Caveat on Accuracy and Precision

In principle, Psychopy can be highly accurate. In practice, much depends on specifics of the experiment and context [GV14][Pla16]. Consider: one study has reported that Galton observed slightly *faster* response times in Victorian times than are observed in contemporary experiments [WTNM13]. Could it be that the Victorians were mentally faster than us? An alternative suggestion for this has been that timings on digital devices are only ever approximations; i.e., many digital devices could not record increments shorter than 100 ms! Even with modern computer technology, the accuracy of stimulus presentation timing is never better than the screen refresh rate. For example, many laptop monitors have refresh rates of 60 Hz. That is, they can at most show a new stimulus 16.5 ms after the previous stimulus, and all stimulus timing intervals will *at best* be multiples of 16.5.

Remember the distinction between accuracy and precision: some of the inaccuracy of stimulus and response time collection will be random jitter. In many cases, this will simply show up as noise in the data (and thus, decrease the power of the experiment). Systematic distortions are not a necessary consequence [VG16]. But other aspects represent an inherent bias. For example, for build-in sound cards, auditory stimulus presentation onset is preceded by a delay. Typically, this delay will be approximately the same on every trial; but it will lead to a systematic underestimation of stimulus onsets.

For experiments requiring extremely precise measurements, it becomes crucial to measure, minimize and account for inaccuracy and bias. For this, external hardware is required; i.e., light- or sound pressure sensitive detectors. (For a cheap solution, the Raspberry Pi mini-computer can easily be extended for this purpose.)

An example experiment

The following section will guide through the programming of a basic experimental paradigm (a false-memory experiment). It will demonstrate Psychopy functionality required to conduct a typical response time or many other types of experiments. The example will be far from the only way to achieve this goal; many other paths are viable. But following it will show many solutions to typical problems during the creation of a psychological experiment.

Alternative software

A range of alternative software could also have been recommended. In particular, OpenSesame is a convenient tool for those who strictly prefer graphical user interfaces; Psychopy's graphical user interface "Builder", as well as the javascript-based tool jsPsych allow conducting online experiments.

OpenSesame

Another powerful option is *OpenSesame* [MathotST12], programmed by Sebastiaan Mathôt. OpenSesame provides a graphical front-end, but also allows directly injecting Python code for fine-tuning. It is recommended for those who prefer a point- and-click, mouse-based approach while still demanding an open-source, reproducible tool.

Going online: surveys on the internet

While we have come quite far since the days of the Pendulum Chronograph, typically, to ensure precise measurements, time-sensitive experiments were still restricted to dedicated lab computers. Recently, javascript-based tools have made it possible to deliver experiments over the internet, and conduct them in a web browser.

Online Experiments with the Psychopy Builder

This option is in fact build into Psychopy, but is not available from the Coder view requires for Python programming. Instead, it must be accessed from the Builder interface. See the Psychopy website for a demonstration of how this functionality can be used.

JsPsych

jsPsych is a javascript library that provides a great package of functions for behavioral experiments. See the jsPsych website.

References

1.4 Analysing Experimental Data

Subchapters

1.4.1 Introduction to Programming with R

Authors Martin Schultze

Citation Schultze, M. (2019). Introduction to Programming with R. In S.E.P. Boettcher, D. Draschkow, J. Sassenhagen & M. Schultze (Eds.). *Scientific Methods for Open Behavioral, Social and Cognitive Sciences*. https://doi.org/10.17605/OSF.IO/HCJG9

Perhaps the question I am confronted with most when teaching statistics courses to psychology undergrads is "Why should I need this, when I become a therapist"? That's a difficult one to answer in a single sentence, but I will give it a try, nonetheless: "Because everything we know in psychology - and the social sciences in general - is connected to empirical data". Well, it was a first try. As the previous sections of this volume have shown you, in scientific psychology we strive to test our claims using data. Testing our claims (ideally) leads to appropriate theories becoming successful and inappropriate theories being discarded. When you are working with patients or clients, would you not rather use techniques that have been proven to be successful in reducing negative symptoms?

Even more baffling to many of you, now reading these words, is the idea of learning a programming language like R in the first year of studying psychology. But the same reasons for learning statistics in general apply to learning R. To move from observations to conclusions requires multiple steps: encoding observations in data, organizing data, describing data, and testing hypotheses using data. The previous section showed you how to collect data in experimental

settings using Python. In this and the following sections we will try to turn the data we gathered into conclusions about our hypotheses by using R - a multi-purpose tool that can be used for almost anything data-related.

Why R?

Many blogs are filled with heaps of reasons to use R over other software that was traditionally more prominent in the behavioral and social sciences. Here's a rundown of the "best of":

1. It's free

This is the obvious one: R costs you nothing to obtain or use. A lot of other software for data analysis requires you to either buy a specific version or rent a yearly license. R, on the other hand, you can simply download, install, and use. R is developed and maintained by **'non-profit foundation<https://www.r-project.org/foundation/>'_**, meaning it is free of commercial interests.

2. It's free

Yes, I've said this already, but this time it has a different meaning. This "free" pertains to the **'definition of free software as proposed by GNU<https://www.gnu.org/philosophy/free-sw.en.html>'_**, which can be boiled down to the phrase "the users have the freedom to run, copy, distribute, study, change and improve the software". This means that R is an open source software and this applies not only to the R-core itself, but also to the packages, which you can obtain via the official repository. This allows everyone to see what is actually happening - e.g. which formulas are used to compute statistics - and catch mistakes in the software before you publish research that is flawed (see [Eklund2016] for what may happen, when software is not open for such checks).

3. Researchers develop their tools in R and make them available as R packages

One important aspect of doing research is using the correct tools. Correct is often understood to simply mean "not wrong", but in science there is a lot of nuance to "not wrong". Some approaches may simply become outdated, because newer alternatives are available or because someone has developed an adapted version which is better suited for your specific case. Currently, many new methodological developments are implemented as R packages and made available for anyone to use. Commercial software is often slower to react, because there is not enough demand for the approach you might need, in order to implement it into a specific software. Thus, R is a more direct channel to current developments in analysis tools. To be able to use newly developed analysis tools comes with a few caveats, of course, because you lose the vetting process performed by commercial providers.

4. You can develop your own approach and implement it in R

A lot of software is limited to what has been implemented by the developers. This is where R can really shine, because it allows you to implement whatever analysis you want, as long as you are able to formulate it in the R language. Because the most basic building blocks of R are simply mathematical operations and relations there is almost no limit to what you can implement. Be aware, that this does not mean that R is the best tool to implement whatever it is, you're trying to implement; it simply means it is a tool that makes it possible at least.

5. The community

There are hundreds of resources and websites containing tutorials, guides, comparisons of approaches, and assistance. Of course, R comes with help-files and examples and there is **'an extensive list of FAQs<https://cran.r-project.org/doc/FAQ/R-FAQ.html>'_**, but as is the case with most programming languages, the premier resource for specific questions is **'stack overflow<https://stackoverflow.com/>'_**. As I am writing this, there are currently 278380 questions tagged "R" on stack overflow, most of them with well-meaning, detailed responses.

But before we can get to experts' opinions on stack overflow, there are some additional resources you can check, if the presentation in this volume leaves you with open questions. Maybe one of the websites that is visited most often in the early stages of learning R is the **'Quick-R website by DataCamp<https://www.statmethods.net/>'_**, where you will get quick answers to some of the early questions in an alternative way to our presentation here. Another way you may want to try to learn the first steps of R is through online courses like **'Harvard's "Statistics and R"<https://online-learning.harvard.edu/course/statistics-and-r>'_** or **'DataCamp's interactive "Introduction to R"<https://www.datacamp.com/courses/free-introduction-to-r>'_**. There are also a lot of other courses out there,

most of which are not free, however. Our hope is, of course, that you will be in no need for such resources once you are done with this and the following sections, but it cannot hurt to have alternatives.

6. Everyone uses it

As, no doubt, your parents have told you on multiple occasions, this is perhaps the single worst argument on the planet to do something: "Would you jump off a bridge, if your friends did it?" Well, in this case, you might want to, because one of the core necessities for open and reproducible science is communicating your work. Not just results and conclusions, but also how you got there. As I stated above, R can be used for basically all steps between assessing data and producing manuscripts and if others use it, it may be necessary for you to be able to read what they wrote, to gain an understanding of how they structured, analyzed, and visualized their data. On the other hand, writing R code also allows other people to retrace the steps you took, because many people can read it. In 2017, Python and R were among the fastest growing programming languages - measured by the number of Stackoverflow views of questions tagged for either one of those languages ([Robinson2017a], [Robinson2017b]). On **'Bob Muenchen's Website r4stats<http://r4stats.com/>'_** you will find a continuously updated Article with current numbers on the popularity of R ([Muenchen2019]), where it is among the most sought after skills in job descriptions and the second most common software cited in scientific articles.

Gathering your tools

Enough chit-chat about the benefits of R - chances are, that if you are still reading at this point, I do not need to convince you any further to use it. So, let us begin by gathering the necessary materials:

The R-Core

The best way to get R is to simply grab it directly from its provider. R itself and most utensils you can add on to it are gathered in what is called CRAN (Comprehensive R Archive Network). For some nice 90s nostalgia you can visit the **'CRAN website<https://cran.r-project.org/>'_** directly, but we also provide short descriptions of how to *Install R on Windows*, *Install R on Mac OS X*, and *Install R on Ubuntu* below. And for those of you who do not want to run the risk of R withdrawal symptoms: **'here's a link to a short description of how to install it on an Android device<https://selbydavid.com/2017/12/29/r-android/>'_**.

Install R on Windows

Installing R on Windows machines is pretty straightforward. The CRAN Website provides you with an executable for the installation of the latest stable R Version, which you can **'download here<https://cran.r-project.org/bin/windows/base/release.htm>'_**. The only thing you have to keep in mind is that R does not perform automatic updates. That's where it becomes a bit tricky: it is advisable to check for a new R version every now and again - a good estimate going by **'the R version history<https://cran.r-project.org/bin/windows/base/old/>'_** is every three months. To update R it is recommended to install a new R version alongside your current version, just in case the new R version broke something that worked before. For some more details on this procedure (and many others), feel free to check the **'R for Windows FAQ<https://cran.r-project.org/bin/windows/base/rw-FAQ.html>'_**

Install R on Mac OS X

Current versions of R are only available for OS X 10.11 (El Capitan) and above. Since this OS is now five years old, the newer versions should cover most users, but if you are among those running an older version of OS X, you will need to install either R Version 3.3.3 (OS X 10.9 and 10.10) or R Version 3.2.1 (OS X 10.6 to 10.8). All three versions can be **'found here<https://cran.r-project.org/bin/macosx/>'_**.

Prior to installing R on OS X 10.8 or above, you will need to install XQuartz. Simply **'download the dmg-file from the XQuartz-Website<https://www.xquartz.org/>'_** and follow the instructions provided in the installer. Afterwards, please restart your computer, before installing R.

To install R after having installed XQuartz, again simply download the **'installer provided by CRAN<https://cran.r-project.org/bin/macosx/>'_** and run it. Should you be asked to install XCode during this process, please do so. As was the case with R for Windows, R does not perform automatic updates, so you should check for a new version every three months or so.

Install R on Ubuntu

R can be installed from the repositories for many Linux distributions. We will cover the case for Ubuntu here, but you can find an online tutorial for installing R on RedHat on **'this blog<https://blog.sellorm.com/2017/11/11/basic-installation-of-r-on-redhat-linux-7/>'_**, for example.

To install R on a Ubuntu machine, you will need sudo-permissions. Because R is part of the Ubuntu repositories, you can simply install it via:

```
sudo apt install r-base
```

However, this will provide you with an outdated version of R in most cases. To obtain the new version of R (and have it update automatically), there are some additional hoops. First, you need to add the necessary GPG key:

```
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys␣
↪E298A3A825C0D65DFD57CBB651716619E084DAB9
```

Then, you need to add the R repository to your sources list. Depending on the Ubuntu release you are running, this may look like this:

```
sudo add-apt-repository 'deb https://cloud.r-project.org/bin/linux/ubuntu bionic-
↪cran35/'
```

if you are running Bionic Beaver, or like this:

```
sudo add-apt-repository 'deb https://cloud.r-project.org/bin/linux/ubuntu cosmic-
↪cran35/'
```

if you are running Cosmic Cuttlefish. If you are running a different release, simply replace the `bionic` or `cosmic` by the name of your version.

Because this changes the `/etc/apt/sources.list` file, you will need to:

```
sudo apt update
```

which may take a few seconds. Afterwards, you can install R using:

```
sudo apt install r-base
```

which should provide you with the current version. In contrast to installing R on Windows or OS X, this will provide you with automatic updates for R.

Running R for the first time

To run R, either open a terminal (for the OS X and Linux users out there) or run the RGUI program you just installed on your Windows machine. You should be greeted by a wall of text, looking something like this:

```
## R version 3.5.3 (2019-03-11) -- "Great Truth"
## Copyright (C) 2019 The R Foundation for Statistical Computing
## Platform: x86_64-pc-linux-gnu (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.
```

There's a few things to pick apart here, so let's start at the top:

- `R Version` obviously states the current version of R you are using, with its release date and nickname. I have tried and tried to figure it out, but, as shared by **'MattBagg on Stackoverflow<https://stackoverflow.com/questions/13478375/is-there-any-authoritative-documentation-on-r-release-nicknames>'_**, there is apparently no system in the nicknames.
- `free software`: we talked about this above - R is free and free, so you may do with it whatever pleases you. When redistributing it, however, you should keep the license in mind.
- `ABSOLUTELY NO WARRANTY`: this is the big reason some companies are still hesitant to use R in high-stakes situations. If your results are wrong because there is an error somewhere in the R-package you are using to perform your analysis, there is no one you can (legally) blame, but yourself for not checking the code thoroughly enough. Now keep in mind, that this is very rare, because most researchers publishing R packages do not just throw any half-baked ideas on CRAN, because their reputations are also tied to their work. The idea is simply, if you want to be sure everything is correct, check for yourself.
- `how to cite R or R packages in publications`: this is the last point I want to highlight. Many people pour years of their lives into making the procedures work that you can then use for free. Please reward their work by citing them correctly, if your are using it. As a matter of fact, let us make this the first R command we perform:

```
citation()
```

```
##
## To cite R in publications use:
##
##   R Core Team (2019). R: A language and environment for
##   statistical computing. R Foundation for Statistical Computing,
##   Vienna, Austria. URL https://www.R-project.org/.
##
## A BibTeX entry for LaTeX users is
##
##   @Manual{,
##     title = {R: A Language and Environment for Statistical Computing},
##     author = {{R Core Team}},
##     organization = {R Foundation for Statistical Computing},
##     address = {Vienna, Austria},
##     year = {2019},
```

(continues on next page)

(continued from previous page)

```
##     url = {https://www.R-project.org/},
##   }
##
## We have invested a lot of time and effort in creating R, please
## cite it when using it for data analysis. See also
## 'citation("pkgname")' for citing R packages.
```

Using the `citation()` function provides you with an overview and a BibTeX source for citing R. If your analysis was performed in R, please use this function to cite it correctly.

RStudio

The official way to interface with R is either via command line (if you are using OS X or Linux) or using the R GUI (if you are using Windows). Both approaches are very limited in their depiction of information and some might even want to call them ugly. This is why there are multiple frontends you can use for R. For those of you, who are already proficient in Emacs, there is **'ESS (Emacs Speaks Statistics)<http://ess.r-project.org/>'_**, which allows you to interact not only with R, but with a lot of other statistical programming languages as well. For those who enjoy a more customizable interface, I would highly recommend **'Atom<https://atom.io/>'_**, which allows you to interface with Python and R in the same environment and comes with integrated git-functionality. **'Here is a quick description of how to get both working in Atom<https://jstaf.github.io/2018/03/25/atom-ide.html>'_**. However, the most widespread IDE for R is, by far, RStudio.

RStudio is a company based in Boston, MA, developing a variety of different products centered around R. Their initial product was the IDE RStudio, which provides a much nicer GUI for R, than the original. The benefit of RStudio over the other possibilities I talked about above is that it is specifically designed for R and all of its little quirks. Thus, it is not a multi-purpose programming tool, but is focused on giving you the easiest and most intuitive way to interact with R, making it a good tool for learning and using R. Beyond that it works identically across all platforms (Windows, OS X, and Linux), making it a good tool for teaching. It also integrates some extensions on R (like R-Markdown for reporting), which we will get into later in this volume.

To install RStudio, simply **'visit its download page<https://www.rstudio.com/products/rstudio/download/#download>'_** and choose the appropriate version for your system. Be aware, that RStudio is simply a frontend and requires you to have installed R as described in the previous section. In contrast to R, RStudio comes with an integrated possibility of updating - this does not update R, however! So you will still need to check for a new version every three months or so, if you are working on a Windows or OS X machine.

Everything we will do in R in the following sections can be done without RStudio, using either just the command-line version of R or any other IDE. Using RStudio is simply a recommendation to ease your way into using R.

When you start up RStudio, the first thing you should do is to open a new R script. You can do this with `Ctrl+Shift+n` (or `Cmd+Shift+n`, if you are using OS X) or via *File* → *New File* → *R Script*. After doing so, your RStudio window should look something like this:

There are four basic panes in this window. In the top left you have the R script you just opened. This is the spot where you can generate your code. Writing the code does not do anything until it is executed. You can run the R code either by clicking the Run-button (in the top right of this pane) or by using `Ctrl+Enter`. For example, typing in `3 + 4` and executing it will send the command (3 + 4) to the console (the pane on the bottom left). Here you should then have:

```
3+4
```

as a mirror of what you executed and

```
## [1] 7
```

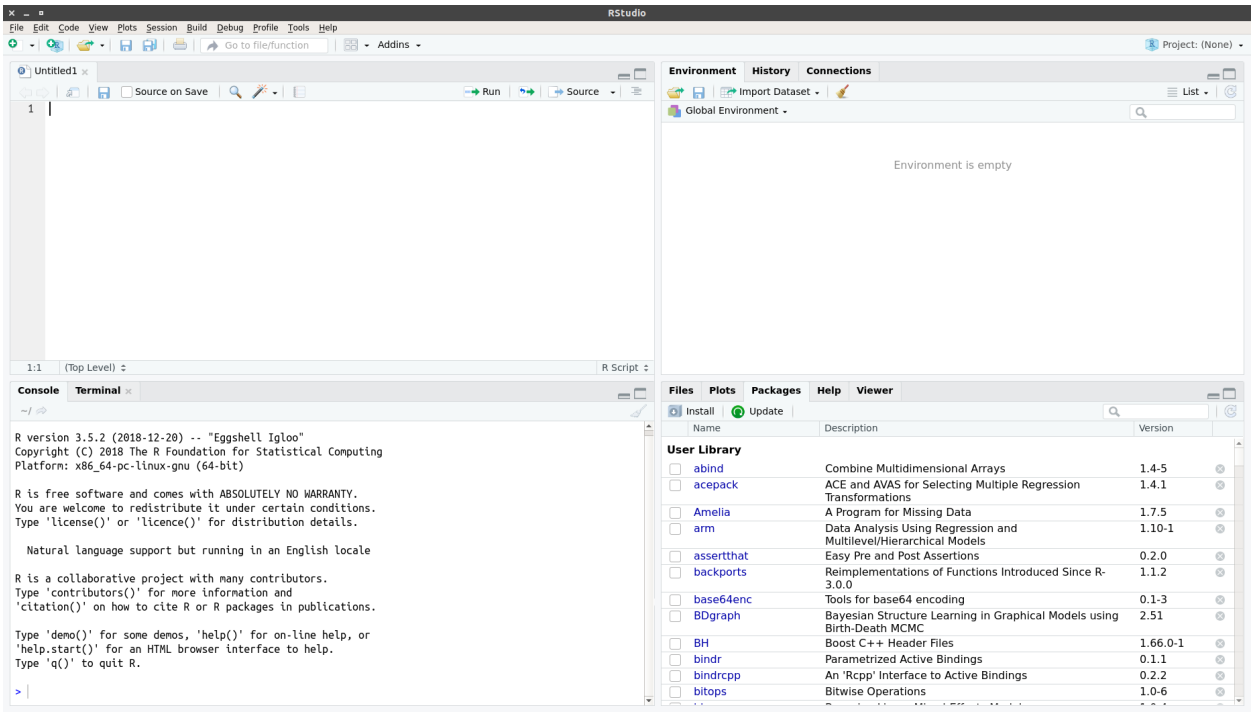

Fig. 2: RStudio just after opening a new R script.

as the result. The layout will be a bit different from what you see on this website: your commands should be preceded by a prompt > and, by default, be in blue, while the result should be in black. Throughout the sections of this volume dealing with R, results will always be preceded by the double hash: ##.

These two panes are what you would find in almost any frontend you could use to interact with R. Where RStudio starts to shine is the remaining two. In the top-right you see a pane labeled "Environment". The other two tabs of this pane are rarely of relevance, so just concentrate on the Environment for now. This pane shows you everything that is currently active in R. We will get into this in a second, but believe me: this makes the first steps in R much easier, because you always have a quick overview of all data you are currently working with. To bottom-right pane has five tabs - all of which are relevant. "Files" gives you the possibility to navigate and open files in R. "Plots" is pretty much self explanatory and we will be generating some nice plots soon. The next tab (the one opened by default) is called "Packages" and gives you an overview of all the extensions for R that are currently installed. You can install new ones and load the ones you have installed from here, but we will be looking at more reproducible way of handling packages in a bit. Perhaps the single most important tab here is the one labeled "Help". Whenever you want to know how a function works, what it does, or how to interpret its output, the help will be opened here.

We have only just started to scrape the surface of what RStudio is and what it can do. If you want some more information on it, the documentation provided by RStudio is great. There are tons of **'webinars for specific topics<https://resources.rstudio.com/webinars>'_**, there is a **'quick overview of how to learn to use RStudio online<https://www.rstudio.com/online-learning/>'_**, and best of all, there are **'a lot cheat-sheets for RStudio and the packages developed by the RStudio team<https://www.rstudio.com/resources/cheatsheets/>'_**. From here on, we will not be focusing on RStudio, but more on the core functionality of R. Feel free to do everything we do in the upcoming sections in RStudio, however.

Some first, wobbly steps

Let's start out with some basics of R code. Because the goal of using R is to write code that leads to reproducible data analysis and results, there are some things you need to know about the general use of R, which we will combine with

some hands-on code writing. So, if you have not opened R yet, it is probably time to do so now.

Commenting and basic functionality

Use comments for everything. I can not stress this enough. Comments are your way of communicating to others and - most often the more important case - to your future self, what you are doing and why. This goes beyond simple small comments and extends to structuring your code. RStudio does a good job of encouraging this, by allowing you to collapse entire sections of your code, if you are currently not interested in looking at it. For the most basic structure, I would recommend using the simple comment character # for small comments and notes. For section titles I recommend beginning the title with #### and ending it with ----. RStudio will automatically recognize this as the section header, but even if you are using something else, this will help you keep your code organized and readable. Let's see how this works with some simple calculations in R:

```
#### Simple calculations ----

3 + 4    # Addition
3 - 4    # Subtraction
3 * 4    # Multiplication
3 / 4    # Division
3 ^ 4    # Powers
```

Here the section is titled "Simple calculations" and each type of calculation is described in a short comment. Now, this may be overkill, but you get the point.

As you can see, I have always left a space between the numbers and the operations. R does not care about empty space. You can even use indentation to help you organize your code without changing the functionality of your code. Beyond this, you do not need to end lines with any specific character - a simple line break ends a line. It is generally recommended to write R as you would write normal sentences, using appropriate spaces to enhance the readability of your code. If you want a detailed style guide for R code, there are **'general recommendations published by Google<https://google.github.io/styleguide/Rguide.xml>'_**.

Now, as we have seen before, executing the basic calculations in your code will result in you receiving a copy of the code you executed, as well as a numeric result in your console. Let's take the division example:

```
3 / 4    # Division
```

```
## [1] 0.75
```

Of course, numeric results are just one kind of result you can obtain from R. As you have seen above, we are often also interested in logical returns. In R, these work something like this:

```
#### Logical relations ----

3 == 4    # Equal?
3 != 4    # Unequal?
3 > 4     # Greater than?
3 < 4     # Smaller than?
3 >= 4    # Greater or equal?
3 <= 4    # Smaller or equal?
```

The first one, as you should expect, returns

```
3 == 4    # Equal?
```

```
## [1] FALSE
```

and the second one returns

```
3 != 4   # Unequal?
```

```
## [1] TRUE
```

Internally, `TRUE` is coded as a 1, while `FALSE` is coded as a 0. Besides making sense, this also results in a lot of nice properties, we will be making use of soon. One quick tip: as with most programming languages the `!` denotes negation in R, so you could also construct a more complicated version of `!=` by hand:

```
!(3 == 4)
```

```
## [1] TRUE
```

which can be read as "not (3 equals 4)". Because the parentheses are evaluated first, they return a `FALSE` and this result is negated by `!`, leading to the final `TRUE`. Why would you ever need this? Well, we will see.

Functions and arguments

What we have looked at so far are simple calculations and equality/inequality checks. These are somewhat special, because the deviate from the "normal way" of doing things in R. Normally, you use functions in R. Using the basic addition shown above, you would write:

```
3 + 4 + 1 + 2
```

```
## [1] 10
```

but the way more akin to how R works in all other instances is by using the `sum` function.

```
sum(3, 4, 1, 2)
```

```
## [1] 10
```

Now, from this simple example you can already derive the basics of how functions work in R. The general structure is always

```
function(argument1, argument2, argument3, ...)
```

As you can see, the name of a function is written first and all the arguments the function requires are passed to it in parentheses, separated by commas. The `sum` function is special in a way, because it can basically take an infinite number of arguments. Let's look at a more prototypical function:

```
log(100)
```

```
## [1] 4.60517
```

As you can see, this returns the natural logarithm of 100. However, what if I want a logarithm with a different base? Because we are using 100 as the example, the simplest logarithm would be of base 10:

```
log(100, 10)
```

```
## [1] 2
```

Let us untangle how this works. Remember your school math: $\log_{\text{base}}$ argument = answer. So, the `log` function takes the argument as its first argument and the base as its second argument. Now, because most people simply cannot remember the correct order of all arguments for the unbelievable number of functions you can use in R, there is a second way of using functions:

```
log(x = 100, base = 10)
```

```
## [1] 2
```

In this approach, you need to name the arguments, but are now free to provide them in any order you wish:

```
log(base = 10, x = 100)
```

```
## [1] 2
```

How can you ever know the names and order of the arguments to a function? There are a few different possibilities, the quickest one is probably:

```
args(log)
```

```
## function (x, base = exp(1))
## NULL
```

So what does this mean? The function `log` has two possible arguments: `x` and `base`. However, what `base = exp(1)` tells you, is that there is a default in place for the `base`-argument. So, if you do not provide a value for this argument, the default is used. In this case e is defined for the exponential function `exp`, but not separately. So if you use the `log`-function, `exp(1)` is evaluated and passed to `log` as an argument. This already shows you that functions can be nested in R: the `exp`-function needs to be evaluated to provide an argument for `log`, so it must be evaluated first. This leads to the same simple principle you find in equations, which can make complicated R code frustrating to read: nested functions are evaluated from the inside out. There are several ways to avoid this, which we will get to in bit.

Getting and using help

While we used `args` to get a quick overview of the arguments for the logarithm in R in the previous section, R actually comes with a very good integrated help system. For any function you know the name of, you can simply use the `help` function. In most cases, this is opened in a new window or pane, which means you can have the help opened at all times. I would encourage you not to be shy about your usage of `help`. It is a much better, efficient way of learning R than typing and retyping arguments over and over. If you are using RStudio, Atom, or something similar, there is also often some form of auto-completion to help you with functions and their arguments. If it is too much effort for you to type `help(function)`, you can also use `?function` to achieve the same result.

So, let's look at the help for the logarithm:

```
help(log)
```

opens up the help file for the `log` function. You can scroll through this help, but here is a short rundown of the basic layout of any R help file:

Description: Usually a very short overview of what the function does.

Usage: The basic structure of the function. This contains all of the arguments you can use. In some cases, like the one you are currently looking at, this may contain multiple functions that are documented together, because they work in the same way. As we discussed above, if an argument is followed by `= something`, it has a fixed default. If you do

not provide a value for that argument, the default is used. Conversely, this also means that any argument not followed by the equals-sign does not have a default an *must* be provided.

Arguments: This shows you a list of all the arguments the function accepts with a brief description of what they do and which format they must adhere to.

Details: Additional information you may want. For functions that perform complex analysis, this may contain a detailed description of the procedure with appropriate citations.

Values: A list of all the output a function produces. In R results of functions are often much bigger than what is printed, when you use them. The values listed here are all the values that are returned, even though you may not see or interact with them on a regular basis.

See Also: If you did not find what you were looking for, maybe these similar functions can help you.

Examples: Maybe the most important section. All documentation in R must ship with minimal working examples. Often the list of arguments can be overwhelming, so you may want to scroll to the bottom to look at the examples in order to see the functions in action. What you can do is copy examples and paste them into your R script to execute them. Basically, this is the same as asking the judges to hear the word in sentence when you were contestant in a spelling bee.

Apropos

`apropos` is function you can use instead of `help`. Using `help` requires you to know the exact, specific name of the function you need help with. Most of the time that is not the situation you need help in. Often the actual name of a function eludes you, which is where `apropos` (or its short version `??`) comes into play.

```
apropos(logarithm)
```

should open a list of some possible functions you could have meant. From here, you can navigate the help files of these functions.

Messages, warnings, and errors

There is no way around it: mistakes happen. When using R, especially in the learning phases, you will produce code that is incorrect, produces errors, or does not do what you expected it to. It is important to know, that this is nothing to be afraid of. One of the advantages of R is that it is made for people who are not professional programmers, so it is rather forgiving in how mistakes can be handled. Because you can execute R code a line at a time, you can avoid the anxious time spent waiting for your code to compile before punishing you with error messages. Instead, you get an immediate feedback on what you did wrong - always think of this, when you are struggling through countless R errors.

On a fundamental level, R has three ways (in addition to just producing correct output) to communicate with you: messages, warnings, and errors.

Messages are simply a sign of a chatty programmer. Often times they provide information about the options with which you invoked a function or tell you about a package being in a beta-State. The startup we looked at in the section *Running R for the first time*, was such a message: it gives you additional information. You can produce messages yourself:

```
message('I am peckish.')
```

```
## I am peckish.
```

This makes sense if you are running long scripts or writing your own functions and want to produce some output to give you a progress update, for example.

The second tier are warnings:

```
warning('I am hungry.')
```

```
## Warning: I am hungry.
```

Warnings indicate that something probably did not go as planned. This means that the function you called still produced output, but you should check to see, whether it is really what you wanted. You can produce a warning for the logarithm-example by

```
log(-1)
```

```
## Warning in log(-1): NaNs produced
```

```
## [1] NaN
```

This still produces output (`NaN`, meaning 'Not a Number'), but tells you that something went awry in a warning message. If you produce a lot of warnings (more than 10, by default) R will simply say something like `There were 11 warnings (use warnings() to see them)`. Then, executing `warnings()` will give you a detailed output about the warnings you produced. If you really produce a lot of warnings (more than 50, by default) R will stop counting them and only return the first 50 when you invoke `warnings()`.

The third tier are errors. Errors mean, that the function you called was aborted and that no output was produced. A typical error is providing the wrong arguments to functions:

```
log(argument = 10)
```

```
## Error in eval(expr, envir, enclos): argument "x" is missing, with no default
```

Just like messages and warnings, you can also produce them yourself

```
stop('I am starving.')
```

```
## Error in eval(expr, envir, enclos): I am starving.
```

Note that errors are produced using the `stop` function, not with a function called `error`. This underlines that the code is stopped at that point. If you are writing a function this means that the execution of the function is aborted at that point and that the error-message you provided is returned. In long scripts this does not mean, that the next line will not be executed, however! Because the next line is a new command, R will simply continue on without having produced the previous results, which can often result in very long chains of errors.

The text produced by warnings and errors is written to be useful in all cases, in which they can occur, so it often does not seem all that helpful. However, once you develop a better understanding of the inner workings of R, you will start noticing that they actually tell you exactly what the problem is.

Objects and the Environment

Now we are really getting into the bread and butter of R. What we saw above - typing in a function and getting a result printed out - is less frequently of interest in R than storing results of a function and using them again in some other fashion. It is also where R gains a leg-up on many of its competitors in the market of data analysis software. Storing results from one type of analysis and then using these as the data for a different type of analysis gives you the flexibility of doing whatever you want with R. Its implementation is also extremely intuitive, so let us take a look:

```
my_num <- sum(3, 4, 1, 2)
```

As you can see, you did not get a result. The result of the `sum` is simply stored in the object called `my_num`. The arrow <- assigns the result of the right side to whatever object is on the left side. This also works in reverse:

```
sum(3, 4, 1, 2) -> my_num
```

but the first version is much more common, because it allows you to see the objects you have created faster. One important thing that just happened, that I want to draw your attention to, is that there was no warning whatsoever. In R objects are simply overwritten if you assign new content to them, so it is best to be very aware of the names for objects that you have already used. This makes it doubly important to use distinctive names for your objects (the other reason being that you want to know what is happening). The **'Google Styleguide for R<https://google.github.io/styleguide/Rguide.xml>'_** that I mentioned above also contains some guidelines on how you should name your objects. These are only guidelines, however, and objects can have any name that does not start in a number.

Now that results are in an object, how do we get access to them? The easiest way is to simply write the name of the object:

```
my_num
```

```
## [1] 10
```

which is shorthand for writing `print(my_num)` or `(my_num)`. But the goal of assigning values and results to objects is to be able to pass them on to other functions. So, in this simple example:

```
sqrt(my_num)
```

```
## [1] 3.162278
```

passes our object to a function. This is essentially the same as:

```
sqrt(sum(3, 4, 1, 2))
```

```
## [1] 3.162278
```

which evaluates the `sum` and then passes its results to `sqrt`. As you have probably guessed, there is no end to the possibilities of nesting functions or creating objects. So

```
my_root <- sqrt(my_num)
```

uses the object `my_num` as an argument in the square-root function and then stores the result in a new object called `my_root`.

Again, *we* decided how to name this object. Instead of naming it `my_num`, we could have named it `cheesecake` or `captain_marvel`. Of course these names would not be very descriptive and would probably confuse us in the future as well as others trying to use the code. If you are using RStudio you have probably realized that both objects have appeared in the Environment tab of top-right pane. RStudio gives you continuous information on what you are currently working with. Any object in the global environment (the one you are currently working in) can be accessed, used, and overwritten. The traditional R way of looking at your environment is

```
ls()
```

```
## [1] "my_num"  "my_root"
```

which lists all objects and functions that you have created. If your workspace has gotten out of hand, you can also list only some objects with

```
ls(pattern = 'num')
```

```
## [1] "my_num"
```

This shows you all objects which contain `num` in their name. Removing objects from your workspace is also quite simple:

```
rm(my_num)
ls()
```

```
## [1] "my_root"
```

Again, notice that you do not get a warning - the object simply disappears - so you might want to be rather careful with using `rm`. If you want everything in your workspace to disappear and start over with a blank slate, you can combine `rm` and `ls`:

```
rm(list = ls())
```

where you simply provide the entire environment (as produced by `ls`) as an argument to `rm`.

Handling data

As you saw in the previous section, objects are where results and numbers are stored. Data you assess is no different, it is only bigger. As discussed in an earlier chapter, variables are the basis of assessing behavior and multiple variables are combined into datasets.

R is extremely rarely used to manually input any data. Most of the time it is either imported from a program you used to assess your experimental data (e.g. from Psychopy), downloaded from a provider you used for assessing data online (from Limesurvey, Unipark, or something similar) or transferred from a different source of data storage (e.g. from an Excel-Sheet). Nevertheless, knowing how data can be created in R can be an incredible help to understanding how data is structured, when it comes from somewhere else.

One more important thing before we continue. In case you were testing all of the previous commands directly in the console, I would like to remind you that we have a script open. This should be used for writing down and commenting the code from this exercise. Do not forget to regularly save it, as you would any other work in progress. You can copy-paste the commands from the following sections into your script, give them a descriptive comment and execute them right from the editor. Just select the row you would like to execute and hit the Run button. You can also use the `Ctrl + Enter` shortcut (`Cmd + Enter` on Macs).

Vectors

So let us build a minimal example: say you observed reaction times of five participants in a **'Stroop test<http://www.yorku.ca/pclassic/Stroop/>'_** ([Stroop1935]), one of the classics of experimental psychology. The basic idea is best conveyed in a picture:

green	purple	blue
purple	red	green

The Stroop effect is the difference between the time it takes you to correctly name the color a word is printed in, when the word and text color match versus when they do not (see [MacLeod1991] for an overview over the first 50 years of its existence). If you want to see how it works, you can check your performance in an online version on **'Open Cognition Lab<http://opencoglab.org/stroop/>'_**, for example.

Now, let's say you measured six reaction times manually, by administering a minimal version of the Stroop to a friend. The times could be (in milliseconds) 597, 1146, 497, 938, 1080, and 1304. To input data as one vector in R, you can use

```
react <- c(597, 1146, 497, 938, 1080, 1304)
```

Calling the `help` function on `c` (as discussed in *Getting and using help*) reveals that it is a basic function to combine all arguments (in this case six reaction times) into a single object. This object is a vector: a one-dimensional array of information, which is of the same type. You can find out what type of vector you just stored your information in in multiple ways. We can use

```
class(react)
```

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

to start, because that provides us with the most basic information about the object `react`: it is a numeric vector. Using

```
str(react)
```

```
##  num [1:6] 597 1146 497 938 1080 ...
```

we obtain a bit more detailed information about the *structure* of the object: it is numeric (`num`), it contains the elements one through six (`[1:6]`), and we see a preview of this object, namely its first five elements.

There are three general types of vectors in R:

Type	Shorthand	Content
`logical`	`logi`	`TRUE` or `FALSE`
`numeric`	`num`	Any type of number
`character`	`chr`	Any combination of letters and numbers

Continuing with the Stroop example, the color of the text that was presented is relevant information. We could encode this in a character vector:

```
color <- c('green', 'purple', 'blue', 'purple', 'red', 'green')
```

We can check whether this is a `character` vector with

```
is.character(color)
```

```
## [1] TRUE
```

In general, the `is.` prefix can be combined with all types of data storage in R, to check whether it is of that type. The same goes for `as.` which can be used for a simple attempt to convert data from one type to another. For the vector-types we have seen, you could use

```
as.numeric(color)
```

```
## Warning: NAs introduced by coercion
```

```
## [1] NA NA NA NA NA NA
```

As you can see, this produces a warning (see *Messages, warnings, and errors*) and the resulting vector contains only `NA`. This is R's way of encoding the absence of information and is short for *not available*. This occurs, because R has no idea how to transform the word `'green'` into a number. Using the basics of measurement theory that were discussed in an earlier chapter, we know that what R is missing is some form of adequate relation. We will discuss how this is done in *Factors*, but for now, let us continue with vectors.

Next to the color, the actual text we are presented with in the Stroop test is also quite important. So, we can generate another character vector:

```
text <- c('green', 'purple', 'blue', 'green', 'blue', 'red')
```

Now, the core effect found by [Stroop1935] is that the reaction is slower, when the color and the text are incongruent. We can use the logical relations shown in *Commenting and basic functionality* to generate a logical vector:

```
cong <- color == text
```

In *Commenting and basic functionality* we saw how comparisons work, when we compare two elements. An incredible positive about R is that most things (e.g. functions and mathematical operations) also work when applied to entire vectors or matrices of data. What happened in this instance, is that the elements in `color` and the elements in `text` were compared one-by-one: is the first element in `color` the same as the first element in `text`? Is the second element in `color` the same as the second element in `text`? And so on... This results in a logical vector of the same length as the two original vectors, because they were compared element-wise:

```
cong
```

```
## [1]  TRUE  TRUE  TRUE FALSE FALSE FALSE
```

As you can see, this is a logical vector:

```
is.logical(cong)
```

```
## [1] TRUE
```

Factors

R's way of storing variables with a nominal or ordinal scale is a type of special vector called a `factor`. These factors have the special property of being numeric while also storing information about what each numeric value means. Take the color variable from our example: we can convert the character vector containing the colors of the presented to a factor by using

```
color_fac <- as.factor(color)
```

and to obtain some overview of what this now looks like:

```
str(color_fac)
```

```
##  Factor w/ 4 levels "blue","green",..: 2 3 1 3 4 2
```

As you can see, this factor contains numeric values (`2 3 1 3 4 2`), but also encodes what each of these numbers mean, by assigning levels. To see all levels of a factor, you can use

```
levels(color_fac)
```

```
## [1] "blue"   "green"  "purple" "red"
```

As you can probably guess, the numeric values are assigned by the way these levels are ordered. Because the original we converted to a factor was a character vector, these levels are ordered alphabetically. Specifically, all unique values of the vector:

```
unique(color)
```

```
## [1] "green"  "purple" "blue"   "red"
```

are ordered and then used as the levels of the factor. Printing the contents of the factor returns the levels, which are associated with each value, not the number that is stored:

```
color_fac
```

```
## [1] green  purple blue   purple red    green
## Levels: blue green purple red
```

which is much more useful, because we will rarely have code-book lying next to our screen where we can look up what each number means. Additionally printing a factor returns the *possible* values, meaning all levels of the factor. Be aware that this makes it possible to have levels of factors, which are not realized in the data.

The dual storage of information makes it, so that factors can easily be converted to `numeric` or `character`:

```
as.numeric(color_fac)
```

```
## [1] 2 3 1 3 4 2
```

```
as.character(color_fac)
```

```
## [1] "green"  "purple" "blue"   "purple" "red"    "green"
```

whichever is more relevant at the moment. However, even though there are numbers associated with each level, the order of the values is arbitrary, meaning normal factors encode nominal scales. You can even change which level comes first, i.e. which level is the reference level, by using:

```
color_fac <- relevel(color_fac, 'green')
```

This command overwrites the original object `color_fac` with a new version, where `'green'` is the first level. All other levels are simply moved back:

```
levels(color_fac)
```

```
## [1] "green"  "blue"   "purple" "red"
```

If your original is a character vector, the strings are simply used as the levels. If your original vector is numeric, this does not really help you. Take the numeric version of our colors:

```
color_num <- c(2, 3, 1, 3, 4, 2)
```

and convert it to a factor:

```
color_fac2 <- as.factor(color_num)
levels(color_fac2)
```

```
## [1] "1" "2" "3" "4"
```

the resulting levels are not really helpful. In this case, you can provide new levels to the object.

```
levels(color_fac2) <- c('blue', 'green', 'purple', 'red')
color_fac2
```

```
## [1] green  purple blue   purple red    green
## Levels: blue green purple red
```

Let's take a quick look at how this works: there are four levels (1, 2, 3, 4) from the conversion of the numeric vector. These four levels can be provided with new labels (blue, green, purple, red). Thus, it is important that there are actually four levels, which we assign to the levels attribute. We don't need to assign the values for each observation of the variable, only the unique levels.

Now, as I've noted, normal factors encode nominal scales. You can also encode ordinal variables with the `ordered` type. Say we ordered the colors by their wavelengths: purple (with the shortest wavelength), blue, green, red.

```
color_ord <- as.ordered(color)
color_ord
```

```
## [1] green  purple blue   purple red    green
## Levels: blue < green < purple < red
```

Well that's not what we wanted. I will leave it up to you to find out how the correct order of colors can be achieved in this case! At this point, all you need, is to be aware that unordered (i.e. nominal) and ordered (i.e. ordinal) variables can both be used in R. As a matter of fact, this is one of the many cases in R, where one is simply a special version of the other:

```
is.factor(color_ord)
```

```
## [1] TRUE
```

```
is.ordered(color_fac)
```

```
## [1] FALSE
```

meaning that `ordered` is a special case of `factor`.

Combining data

As a result of the section on *Vectors*, we have four different objects in our environment, which all relate to the same thing. Naturally, the best idea would be to combine them somehow. As with vectors, there are multiple types of storing data sets in R, but their relationships are a bit more complicated. Let's get a general overview:

Type	Content
`matrix`	Vectors of the same length and type (two dimensional)
`array`	Vectors of the same length and type (n-dimensional)
`data.frame`	Vectors of the same length
`list`	Any objects

As you can see, the types are more specialized the further to they are to the top of the table. More specialized types restrict your possibilities of combining arbitrary information, but make storing and handling data more efficient in terms of computational power. Especially when handling abstrusely large data (such as raw fMRI or genetical data), I would highly recommend using matrices. Matrices are especially useful, because you can simply apply matrix-algebra to them, making computation and data analysis much easier.

As you can probably tell from the table, a `matrix` is a special case of an `array` - the two dimensional one. Less obvious is the fact that :code:‘data.frame‘s are special cases of :code:‘list‘s, i.e. the one where all content is of exactly the same length.

Let's begin by constructing a matrix. For this, we need to ensure that the objects we intend to combine are of the same type and of the same length:

```
class(color)
```

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

```
class(text)
```

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

```
length(color)
```

```
## [1] 6
```

```
length(text)
```

```
## [1] 6
```

or, more simply:

```
class(color) == class(text)
```

```
## [1] TRUE
```

```
length(color) == length(text)
```

```
## [1] TRUE
```

If we want to combine these two to a matrix, there are multiple ways, but the two main approaches are, by either using the `matrix` function or by using `cbind`. We will use the second approach here, but I encourage you to take a look at `help(matrix)` and try this approach to reconstruct what is happening here.

The function `cbind` refers to *binding* vectors together as multiple *columns*. Traditionally, data frames are organized in such a fashion, that columns represent different variables, while rows represent different observations (e.g. people). If you wanted to combine data from different people that were observed on the same number of variables (e.g. the six reaction times of two different people) you would use `rbind`, for *binding* multiple *rows*. In our case, we can combine `text` and `color` to a matrix:

```
mat <- cbind(color, text)
```

The resulting object is a matrix:

```
class(mat)
```

```
## [1] "matrix"
```

but - because matrices are special cases of arrays - it is also an array!

```
is.array(mat)
```

```
## [1] TRUE
```

What matrices are not, is special cases of :code:‘data.frame‘s or :code:‘list‘s:

```
is.data.frame(mat)
```

```
## [1] FALSE
```

```
is.list(mat)
```

```
## [1] FALSE
```

Combining `color` and `text` worked, because both are of the same type (`character`). However, the data we have is also numeric (the reaction times) and logical (the indicator of congruence). If you combine all of them using the `cbind` command, the following will happen:

```
mat <- cbind(color, text, cong, react)
mat
```

```
##      color    text     cong    react
## [1,] "green"  "green"  "TRUE"  "597"
## [2,] "purple" "purple" "TRUE"  "1146"
## [3,] "blue"   "blue"   "TRUE"  "497"
## [4,] "purple" "green"  "FALSE" "938"
## [5,] "red"    "blue"   "FALSE" "1080"
## [6,] "green"  "red"    "FALSE" "1304"
```

All vectors were combined, but they were all converted to the most general type of vector of the three: `character`. This is bad, because you loose the numeric information in the variable `react` and can not use it for calculations and statistical analysis.

This is why, in most cases you will encounter with behavioral data, :code:‘data.frame‘s are the type of storage needed. You can combine the four vectors like this:

```
dat <- data.frame(color, text, cong, react)
```

This results in a `data.frame` with six rows and four columns. You can check this with the specific functions `nrow` and `ncol`, or get a general overview with:

```
str(dat)
```

```
## 'data.frame':    6 obs. of  4 variables:
##  $ color: Factor w/ 4 levels "blue","green",..: 2 3 1 3 4 2
##  $ text : Factor w/ 4 levels "blue","green",..: 2 3 1 2 1 4
##  $ cong : logi  TRUE TRUE TRUE FALSE FALSE FALSE
##  $ react: num  597 1146 497 938 1080 ...
```

```
dat
```

```
##    color   text  cong react
## 1  green  green  TRUE   597
## 2 purple purple  TRUE  1146
## 3   blue   blue  TRUE   497
## 4 purple  green FALSE   938
## 5    red   blue FALSE  1080
## 6  green    red FALSE  1304
```

As you can see, R automatically converts character vectors to factors! This is because that is what is most often desired. As with (almost) all behavior of R, you can adjust this. As we saw in *Functions and arguments*, this is only a matter of identifying the correct argument and changing its value. You can check `help(data.frame)` and will see that the argument we are looking for is aptly named `stringsAsFactors`. So:

```
dat2 <- data.frame(color, text, cong, react, stringsAsFactors = FALSE)
```

will provide us with a `data.frame` in which the character vectors remain as such. We can check:

```
str(dat2)
```

```
## 'data.frame':    6 obs. of  4 variables:
##  $ color: chr  "green" "purple" "blue" "purple" ...
##  $ text : chr  "green" "purple" "blue" "green" ...
##  $ cong : logi  TRUE TRUE TRUE FALSE FALSE FALSE
##  $ react: num  597 1146 497 938 1080 ...
```

The three types discussed so far all assume that the vectors we combine are of the same length. What happens when they are not? Let's generate a vector with five entries. Because we have not particular data for this example, we can just fill it with a sequence from 1 though 5.

```
five <- 1:5
five
```

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

In this case the `:` is a shorthand for `seq(1, 5, 1)`, meaning a sequence is generated from 1 through 5 in steps of 1. With the `seq` function you can generate all kinds of sequences - feel free to check `help(seq)`.

Combining this five-entry vector with our other variables results in an error:

```
data.frame(color, text, cong, react, five)
```

```
## Error in data.frame(color, text, cong, react, five): arguments imply differing␣
↪number of rows: 6, 5
```

which shows you that `data.frame`s need all their variables to be of the same length. This makes sense, when you think about what the data represents: usually each row of a data set is a person or trial, why would some trials have less variables than others? But, say the reaction timed out for the sixth trial, this does not result in a shorter vector, but simply in that instance being a missing value - :code:`NA` in R verbiage. You can achieve this by:

```
five <- c(five, NA)
```

NA`s can be used in any type of vector - they do not change the type of vector, they simply represent the absence of information. This turns the vector into a vector with six entries, the last of which is :code:`NA. If you are adding a vector to a `data.frame`, you do not need to enter all vectors, by the way. You can add a vector to an already existing `data.frame`:

```
data.frame(dat, five)
```

```
##    color   text  cong react five
## 1  green  green  TRUE   597    1
## 2 purple purple  TRUE  1146    2
## 3   blue   blue  TRUE   497    3
## 4 purple  green FALSE   938    4
## 5    red   blue FALSE  1080    5
## 6  green    red FALSE  1304   NA
```

One final word of caution: in R there is a special exception to the "must be of the same length"-rule. An exception is made when the shorter vector is a divisor of the longer vector. In that instance, the shorter vector is repeated until the data is filled. Let's take the vector of 1 through 3 as an example:

```
three <- 1:3
data.frame(color, text, cong, react, three)
```

```
##    color   text  cong react three
## 1  green  green  TRUE   597     1
## 2 purple purple  TRUE  1146     2
## 3   blue   blue  TRUE   497     3
## 4 purple  green FALSE   938     1
## 5    red   blue FALSE  1080     2
## 6  green    red FALSE  1304     3
```

so you will need be careful when adding new variables: always check whether the new data is actually what you intended.

The final way of storing data is simultaneously the least efficient and most regularly used form: `lists`. The latter is the case because most functions return lists as results. For very large data sets I would advise against using list, because they tend to slow everything down quite drastically. In general, if it is at all possible to simplify your data into a data type that is above it in the table I presented at the beginning of this section, you should probably do it.

Nevertheless, lists are useful, because you can combine all types of information and data. A simple case is a list of different vectors:

```
lst <- list(color, text, cong, react)
str(lst)
```

```
## List of 4
##  $ : chr [1:6] "green" "purple" "blue" "purple" ...
##  $ : chr [1:6] "green" "purple" "blue" "green" ...
##  $ : logi [1:6] TRUE TRUE TRUE FALSE FALSE FALSE
##  $ : num [1:6] 597 1146 497 938 1080 ...
```

The structure of this looks eerily similar to that of the `data.frame` we looked at before. That is because, as mentioned, :code:'data.frame's are simply special lists. The difference is that you can store anything in your list, even other lists!

```
meta_list <- list('Person 1', lst, dat)
str(meta_list)
```

```
## List of 3
##  $ : chr "Person 1"
##  $ :List of 4
##   ..$ : chr [1:6] "green" "purple" "blue" "purple" ...
##   ..$ : chr [1:6] "green" "purple" "blue" "green" ...
##   ..$ : logi [1:6] TRUE TRUE TRUE FALSE FALSE FALSE
##   ..$ : num [1:6] 597 1146 497 938 1080 ...
##  $ :'data.frame':        6 obs. of  4 variables:
##   ..$ color: Factor w/ 4 levels "blue","green",..: 2 3 1 3 4 2
##   ..$ text : Factor w/ 4 levels "blue","green",..: 2 3 1 2 1 4
##   ..$ cong : logi [1:6] TRUE TRUE TRUE FALSE FALSE FALSE
##   ..$ react: num [1:6] 597 1146 497 938 1080 ...
```

In many cases, the results of functions are rather complicated lists. For example, the result of a regression in R is a list of 13 elements of various types and sizes, so it is useful to know how to interact with lists, even if your own data should ideally be stored in a different format.

Extracting data

In the previous two sections the focus was on combining data into larger objects. While this is normally what you do when gathering data, inspecting specific information is just as important, especially because, as noted above, results that are output by analysis functions are often lists.

Let us start with the simplest case: extracting an element from a vector. The four vectors we generated in the section *Vectors* all contain six elements. Take a closer look at the structure of the reaction times:

```
str(react)
```

```
##  num [1:6] 597 1146 497 938 1080 ...
```

The `[1:6]` tells you that this vector contains elements one through six. The brackets indicate how to subset these elements. For example, if you want to see the fourth element of this vector:

```
react[4]
```

```
## [1] 938
```

This returns the fourth element. In R the brackets `[ ]` are the most basic way of selecting specific elements in any object. What you write in those brackets then determines what you select. You can also explicitly deselect something that is not of interest to you:

```
react[-4]
```

```
## [1]  597 1146  497 1080 1304
```

The important thing to keep in mind here, is that this selection works, like most things in R, for vectors just as well as it does for single elements. So creating a selection vector can help:

```
sel <- c(1, 3, 5)
react[sel]
```

```
## [1]  597  497 1080
```

of course, you do not need to create an object for the selection vector, you can pass it directly (i.e. `react[c(1, 3, 5)]`) and it will have the same effect. This works according to the same principle we discussed in *Functions and*

arguments: functions can be nested in functions and, because they are evaluated from the inside out, their results will be used as the argument. In this case the `c` function is evaluated and its result (the vector) is passed to the brackets. In case you were wondering: you can also use this to select the same element multiple times.

```
react[c(1, 1, 2)]
```

```
## [1]  597  597 1146
```

The selection we performed up until here was based on the numeric representation of an element's position in a vector. You can also use `character` and `logical` vectors to select elements. We will see how this works for `character` vectors in a second, but the logical vector provides an immense amount of flexibility. Recall the vector we constructed to indicate, whether text and color are the same (i.e. whether it is a congruent condition). We can now use this vector to logically select the elements of any other vector that is also six elements long. So, to select the reaction times for congruent conditions:

```
react[cong]
```

```
## [1]  597 1146  497
```

For the incongruent conditions, we can simply use the `!` to negate the logical vector:

```
react[!cong]
```

```
## [1]  938 1080 1304
```

Because vectors are one-dimensional, selecting elements from them requires only one information. Most data are stored in two (or more) dimensional objects, however. As shown in *Combining data*, there are four standard variants of data storage. First, let's look at the matrix of colors and texts:

```
mat <- cbind(color, text)
mat
```

```
##      color    text
## [1,] "green"  "green"
## [2,] "purple" "purple"
## [3,] "blue"   "blue"
## [4,] "purple" "green"
## [5,] "red"    "blue"
## [6,] "green"  "red"
```

This matrix has six rows and two columns, so to select any single element we need to locate it along these two dimensions. So to select the "red" text, we need to navigate to the 6th row and the 2nd column:

```
mat[6, 2]
```

```
##  text
## "red"
```

In R, as is usual, the first dimension of matrices is the row and the second dimension is the columns. Thus, the brackets we use as our "selection function", now take two arguments. If we want all elements along one dimension we can use:

```
mat[1, ]     # All elements of the 1st row
```

```
##   color    text
## "green" "green"
```

```
mat[, 1]      # All elements of the 1st column
```

```
## [1] "green"  "purple" "blue"   "purple" "red"    "green"
```

It is possible to select elements in matrices by using a one-dimensional notation: `mat[7]` will return the seventh overall element, first counting through all rows of the first column, then continuing with second column and so on. However, I would strongly advise you to use the two-dimensional version of selecting data from matrices for now.

The two-dimensional selection procedure is, of course, extendable to arrays of more than two dimensions, where you simply provide more arguments to the brackets (e.g. `[3, 1, 4]` will select the 4th row, 1st column, 4th layer). If you want to select more than one element, you would need to provide vectors determining your selection along on dimension. So, let's say you want the 1st and 4th element of the 1st column:

```
mat[c(1, 4), 1]
```

```
## [1] "green"  "purple"
```

Again, this selection procedure is not limited to numeric vectors, but can also be performed using logical or character vectors. As an example for the use of logical vectors, we can select all rows of the matrix, which are congruent conditions:

```
mat[cong, ]
```

```
##      color    text
## [1,] "green"  "green"
## [2,] "purple" "purple"
## [3,] "blue"   "blue"
```

Character vectors can be used for selection, if the dimensions of a matrix have names. Let's check:

```
dimnames(mat)
```

```
## [[1]]
## NULL
##
## [[2]]
## [1] "color" "text"
```

This is actually the first time we see a function returning a list! The information we can glean from it is that the first dimension (the rows) has `NULL` names - so no names here. The second dimension (the columns) contains the names color and text. These names were simply inherited from the names of the objects we used `cbind` on.

So, knowing the names, we can use a character vector to select columns from this matrix:

```
mat[, 'color']
```

```
## [1] "green"  "purple" "blue"   "purple" "red"    "green"
```

As we discussed in *Combining data*, matrices only work in a limited number of situations and `data.frame`s are much more widespread. Then why did we just spend so much time on matrices? Well, because a :code:`data.frame` is just as two-dimensional as a matrix, so the same procedures we discussed for matrices also work for them.

```
dat[1, 4]        # 1st row, 4th column
```

```
## [1] 597
```

```
dat[, 3]          # All of the 3rd column
```

```
## [1]  TRUE  TRUE  TRUE FALSE FALSE FALSE
```

```
dat[c(2, 3), 3]  # Elements 2 and 3 of the 3rd column
```

```
## [1] TRUE TRUE
```

```
dat[cong, ]       # Only rows for which cong == TRUE
```

```
##    color   text cong react
## 1  green  green TRUE   597
## 2 purple purple TRUE  1146
## 3   blue   blue TRUE   497
```

Remember, that this can be combined with the creation of objects discussed. So, if you want to perform some analyses separately for congruent and incongruent stimuli, you can just create two new :code:‘data.frame‘s, which contain only parts of the originals:

```
con <- dat[cong, ]
inc <- dat[!cong, ]
```

This has the potential to make it much easier to handle subsets of data, when you perform a lot of analyses on certain parts, but not others.

Next to the procedures for matrices, the procedures for lists also work on `data.frame`s, because they are very specific types of lists. The most prominent way of selection from :code:`data.frame`s is by using the :code:`$`:

```
dat$react
```

```
## [1]  597 1146  497  938 1080 1304
```

This type of selection can be read as “in `dat`, select variable `react`”. This is used in extremely many R scripts, because this allows us to store combined data and then quickly pick a single variable for which we want to calculate some statistics. This procedure also works for lists and can even be extended to cases where multiple `$` are used in sequence. Say you have multiple data frames in a list. In that situation you can select a variable in a data frame in a list by `list_name$data_name$variable_name`. However, keep in mind, that for the `$` approach to work, all elements you are trying to select must be named! For `data.frame`s, R generates names automatically (:code:`V1`, `V2`, and so on, if you do not provide names), but this is not the case for regular lists.

To see the names of the variables in your data set, you can simply use:

```
names(dat)
```

```
## [1] "color" "text"  "cong"  "react"
```

If you are more comfortable with using functions instead of the brackets or the `$`, you can also use the `subset` function, which allows you to achieve the same results. If you are interested in seeing how that function works, I encourage you to take a look at `help(subset)`.

Adding new data

We have now seen how to combine separate objects to one data set and how to select and extract specific information from those data sets. The last step is to add new information to already existing data.

As we have seen above, we can use `cbind` or `rbind` to combine multiple vectors either by column or by row. This also works for adding columns and rows to pre-existing matrices. Additionally, we already saw that the `data.frame` function can be used either to combine vectors into multiple columns of a **:cdoe:'data.frame'** or to add vectors to existing ones.

You can also use the approaches for identifying single rows and columns to add new data. Perhaps the most common scenario is adding new variables to a data set. Say we want to use the deviation of a reaction time from a person's normal reaction time in our analyses, instead of the raw time we measured. This could have the advantage of controlling for person-specific variables that influence the overall reaction time, but not deviations in our experiment. As such a baseline we can use the arithmetic mean of the reaction times as a placeholder for a person's average reaction time:

```
mean(dat$react)
```

```
## [1] 927
```

So, to compute the deviation from the mean on each reaction time:

```
dat$react - mean(dat$react)
```

```
## [1] -330  219 -430   11  153  377
```

This is again a vector of 6. We could create an object by using the `<-`, but that would not add the variable to the `data.frame`. It would simply become a free floating object in our environment. To keep data organized, it is better to store them together in a single object.

There are now multiple ways to achieve the goal of adding a new variable to the `data.frame`, but the one you will probably encounter most often is by using the `$` notation. Let's say the new variable is supposed to be called `dif`, for difference. Let's see whether this variable already exists in the data set:

```
dat$dif
```

```
## NULL
```

As you may have guessed, it doesn't. This means, we can simply create it by assigning values to it.

```
dat$dif <- dat$react - mean(dat$react)
```

This works just like it does for creating objects in the environment. If we use a name that does not exist, the variable is created. If we use a name of a variable that exists within the `data.frame`, it is overwritten without warning.

This variable now exists only in the data set. It does not exist freely in the environment:

```
dif
```

```
## Error in eval(expr, envir, enclos): object 'dif' not found
```

This is especially useful, when you are handling many data sets simultaneously, because it can help you avoid overwriting information you may have needed. Instead a variable is assigned straight to the `data.frame` it is related to.

Adding a new variable this way has the benefit of it being named in the process. You could also use the bracket notation to get this done. In our case the `data.frame` now consists of 5 columns:

```
ncol(dat)
```

```
## [1] 5
```

So, if we were to add a new variable it would be most logical to add it as the 6th, or generally speaking, the `ncol(dat) + 1` column:

```
dat[, ncol(dat) + 1] <- dat$react - mean(dat$react)
dat
```

```
##    color   text  cong react  dif   V6
## 1  green  green  TRUE   597 -330 -330
## 2 purple purple  TRUE  1146  219  219
## 3   blue   blue  TRUE   497 -430 -430
## 4 purple  green FALSE   938   11   11
## 5    red   blue FALSE  1080  153  153
## 6  green    red FALSE  1304  377  377
```

This adds the variable in the appropriate spot, but does not provide it with a name. Instead it gets the generic name V6. You could then name it manually, by assigning a name via the `names` function. Because the result of `names` is a vector, you can assign the name for this specific variable, by selecting the appropriate element:

```
names(dat)[ncol(dat)] <- 'dif2'
names(dat)
```

```
## [1] "color" "text"  "cong"  "react" "dif"   "dif2"
```

Because this variable is the same as the variable we added previously, it makes no sense to keep both of them. To remove a variable from a `data.frame`, it needs to be overwritten with nothing. As you may have noticed, R represents nothing with `NULL`:

```
dat$dif2 <- NULL
dat
```

```
##    color   text  cong react  dif
## 1  green  green  TRUE   597 -330
## 2 purple purple  TRUE  1146  219
## 3   blue   blue  TRUE   497 -430
## 4 purple  green FALSE   938   11
## 5    red   blue FALSE  1080  153
## 6  green    red FALSE  1304  377
```

Using the bracket approach, we can also add new rows to our `data.frame`. The only thing we need to keep in mind here, is that the vectors must be constructed correctly, for the `data.frame` to accept them. First, let's remove the new `dif` variable, so we can return to our original `data.frame`:

```
dat$dif <- NULL
str(dat)
```

```
## 'data.frame':    6 obs. of  4 variables:
##  $ color: Factor w/ 4 levels "blue","green",..: 2 3 1 3 4 2
##  $ text : Factor w/ 4 levels "blue","green",..: 2 3 1 2 1 4
##  $ cong : logi  TRUE TRUE TRUE FALSE FALSE FALSE
##  $ react: num  597 1146 497 938 1080 ...
```

this `data.frame` now consists of our four original variables. To add a new row, much like we added a new column, we can simply assign values to the `nrow(dat) + 1` row:

```
dat[nrow(dat) + 1, ] <- c('red', 'red', TRUE, 627)
dat
```

```
##    color   text  cong react
## 1  green  green  TRUE   597
## 2 purple purple  TRUE  1146
## 3   blue   blue  TRUE   497
## 4 purple  green FALSE   938
## 5    red   blue FALSE  1080
## 6  green    red FALSE  1304
## 7    red    red  TRUE   627
```

Be aware that we are only allowed to add rows that fulfill all the requirements of our `data.frame`: they must be of the correct length (i.e. the number of columns of the `data.frame`) and the values in each spot must be compatible with the variables. Factors generally prove most problematic in such situations:

```
dat[nrow(dat) + 1, ] <- c('orange', 'purple', FALSE, 844)
```

```
## Warning in `[<-.factor`(`*tmp*`, iseq, value = "orange"): invalid factor
## level, NA generated
```

```
dat
```

```
##    color   text  cong react
## 1  green  green  TRUE   597
## 2 purple purple  TRUE  1146
## 3   blue   blue  TRUE   497
## 4 purple  green FALSE   938
## 5    red   blue FALSE  1080
## 6  green    red FALSE  1304
## 7    red    red  TRUE   627
## 8   <NA> purple FALSE   844
```

Factors in `data.frame`s only accept values with are contained in their levels. If we want to add the last row we first need to add :code:'orange'' as a level of `color`:

```
levels(dat$color) <- c(levels(dat$color), 'orange')
levels(dat$color)
```

```
## [1] "blue"   "green"  "purple" "red"    "orange"
```

```
dat[nrow(dat), ] <- c('orange', 'purple', FALSE, 844)
dat
```

```
##    color   text  cong react
## 1  green  green  TRUE   597
## 2 purple purple  TRUE  1146
## 3   blue   blue  TRUE   497
## 4 purple  green FALSE   938
## 5    red   blue FALSE  1080
## 6  green    red FALSE  1304
```

(continues on next page)

(continued from previous page)

```
## 7    red    red  TRUE   627
## 8 orange purple FALSE   844
```

Getting data into and out of R

Up until this point, everything we have discussed was handled by R internally. If you followed along with all the examples your workspace should look something like this:

```
ls()
```

```
##  [1] "color"      "color_fac"  "color_fac2" "color_num"  "color_ord"
##  [6] "con"        "cong"       "dat"        "dat2"       "five"
## [11] "inc"        "lst"        "mat"        "meta_list"  "react"
## [16] "sel"        "text"       "three"
```

Most of it is junk that we don't need, but we may want to keep the data frame containing the results of the Stroop trials.

In all functions which relate to loading, importing, saving, and exporting data, R requires filepaths to be specified. This means that you would need to determine the entire path in an absolute sense, every time you interact with a file. This can get quite annoying, so there is a specific way, R handles relative paths, which is called the working directory. (For those of you familiar with the terminal in Mac OS X and Linux or the command line in Windows, this is the same way directories are handled in those.) This is simply the directory you are currently working in - if you want to save or load any file in this specific directory, you can simply provide the filename. To see what your current working directory is, you can use

```
getwd()
```

```
## [1] "/home/martin/smobsc/docs/chapter_ana"
```

Of course, your current working directory will have a different name. Generally, I would recommend setting up a folder for each project you are working on and then using that folder as your working directory. This saves you the time of typing in absolute paths and prevents you from accidentally storing files somewhere, where you need to look for them or accidentally overwrite something (again, R will not warn you, if a file already exists). You can set your working directory by using

```
setwd('PATH/TO/YOUR/DIRECTORY')
```

In the easiest case you can simply navigate to your folder using your file system and copy its location (listed in its properties). Windows users should be aware: Windows uses backslashes `\` to denote subfolders, while R uses forwardslashes :code:`/`. *So, if you copy the folder location on a Windows machine you will need to replace all the :code:`* with `/` in your filepaths.

If you want to see which files are in your working directory, you can use

```
dir()
```

to check. In most cases I highly recommend having an additional subfolder called "data" in the folder for your project. Then, you can use `./data/` to save and load from there.

Saving and loading

R uses two own data formats - RDA and RDS - which you can use to save data to and load data from. Here's a quick overview:

Data format	RDA	RDS
File extension	.rda, .RData	.rds
Save function	`save`	`saveRDS`
Load function	`load`	`readRDS`
Objects saved	Multiple	Single
Loading behavior	Restore objects in environment	Assign loaded data to new object

Generally speaking, RDA is best suited when you are storing multiple objects simultaneously and you do not need them separately. RDS on the other hand is best suited for use with single objects. Most people use RDA regardless of whether they are storing multiple objects or single objects, but we will look at both, because using RDS can be extremely beneficial when using the same routines multiple times - for example if you have a single data frame for each test subject.

First, let's look at how saving and loading RDAs works. For this example, we will save only the data frame `dat` in our "data" subfolder:

```
save(dat, file = './data/dat.rda')
```

Now that we have saved what is relevant to us, let's clear the entire environment, so we can be sure that loading actually loads a file and we are not just seeing the object we already had in our environment:

```
rm(list = ls())
ls()
```

```
## character(0)
```

Our environment is empty now. So, as stated above, loading the RDA should restore the object `dat` in our environment:

```
load('./data/dat.rda')
ls()
```

```
## [1] "dat"
```

So that is all it takes to save and load in R. Let's take a quick look at the alternative: RDS.

```
saveRDS(dat, './data/dat.rds')
rm(list = ls())
```

Loading an RDS requires you to assign the result to an object. This has the benefit that it allows you to use object names that are specific to the script you are working with to analyze your data, not to the one creating it.

```
stroop <- readRDS('./data/dat.rds')
stroop
```

```
##    color   text  cong react
## 1  green  green  TRUE   597
## 2 purple purple  TRUE  1146
## 3   blue   blue  TRUE   497
```

(continues on next page)

(continued from previous page)

```
## 4 purple  green FALSE   938
## 5    red   blue FALSE  1080
## 6  green    red FALSE  1304
## 7    red    red  TRUE   627
## 8 orange purple FALSE   844
```

In general, I would recommend using RDS whenever possible. You might be wondering: if RDA can save multiple objects at once, why not save the entire environment? This is actually what RStudio asks you to do, when you close it. Don't. The core idea of using a programming language like R is that you can use the script to recreate everything that was done. Relying on objects in your workspace that you cannot recreate using your script means that your data preparation and analyses cannot be reconstructed by anyone else.

Importing and exporting

Most of the programs used to assess data do not produce RDA or RDS data. Most often what this means is that you will have to import and clean data, then save it as RDA or RDS, and use those files for your analysis scripts. For a quick glance of how to import from the data-formats provided by SPSS, SAS, STATA, and so on, you can take a look at the overview provided on the **'Quick-R website<https://www.statmethods.net/input/importingdata.html>'_**. In this section we will take a more in-depth look at importing from text-formats, because they are also often what we get from Python-based assessments.

The text-formats you will see most often are .csv, .txt, and .dat. While there is a specific function for .csv files (take a look at `help(read.csv)` if you are interested), the general function `read.table` can be used for all three types, so it is the best one to discuss here.

The experiment that was discussed in the previous sections results in multiple .csv files - one for each participant. We will take a look at how to efficiently extract all of them in just a few lines of code in the next chapter, here we will concentrate on importing one of them.

The `read.table` command is the first one we are looking at, that requires quite a few arguments to get it to do what we want. Let's take a look at all the arguments (you could also use `help(read.table)`):

```
args(read.table)
```

```
## function (file, header = FALSE, sep = "", quote = "\"'", dec = ".",
##     numerals = c("allow.loss", "warn.loss", "no.loss"), row.names,
##     col.names, as.is = !stringsAsFactors, na.strings = "NA",
##     colClasses = NA, nrows = -1, skip = 0, check.names = TRUE,
##     fill = !blank.lines.skip, strip.white = FALSE, blank.lines.skip = TRUE,
##     comment.char = "#", allowEscapes = FALSE, flush = FALSE,
##     stringsAsFactors = default.stringsAsFactors(), fileEncoding = "",
##     encoding = "unknown", text, skipNul = FALSE)
## NULL
```

To know which settings to use, we need to know what our data files look like. Take a look at the first file, the contents should look something like this:

```
## text,is_lure,key,yes_key,rt,subject
## Bier,neutral,l,a,2.1808581352200003,0
## Dinosaurier,neutral,l,a,0.9811301231380001,0
## Volkswagen,correct,a,a,1.0397160053299999,0
## Sibirien,neutral,l,a,2.89859485626,0
```

We can see that the first line in our data file contains the variable names, meaning we have to set `header = TRUE` in our `read.table`. Additionally, the variables are separated by commas, meaning we should use `sep = ','`. Let's

see where this leaves us:

```
drm0 <- read.table('./data/0_recognized.csv',
  header = TRUE, sep = ',')
str(drm0)
```

```
## 'data.frame':    27 obs. of  6 variables:
##  $ text   : Factor w/ 27 levels "Achtung","Auto",..: 4 7 25 24 12 5 19 16 10 8 ...
##  $ is_lure: Factor w/ 3 levels "correct","lure",..: 3 3 1 3 1 1 3 1 3 3 ...
##  $ key    : Factor w/ 2 levels "a","l": 2 2 1 2 1 1 2 1 2 2 ...
##  $ yes_key: Factor w/ 1 level "a": 1 1 1 1 1 1 1 1 1 1 ...
##  $ rt     : num  2.181 0.981 1.04 2.899 2.87 ...
##  $ subject: int  0 0 0 0 0 0 0 0 0 0 ...
```

This seems to have done what we wanted. Remember, there is no problem with simply trying things, running your code, to see what happens. As I stated above, it is one of the main benefits of using R: you do not have to compile your code.

We looked at how you can save this file in an RDA or RDS format in the last section. You can also export it to text-formats to be able to use it in other software. To export, you can use `write.table`, which works very similar to `read.table`:

```
args(write.table)
```

```
## function (x, file = "", append = FALSE, quote = TRUE, sep = " ",
##     eol = "\n", na = "NA", dec = ".", row.names = TRUE, col.names = TRUE,
##     qmethod = c("escape", "double"), fileEncoding = "")
## NULL
```

The first argument that is required is the object we want to export. In our case, this is `drm0`. The second is, of course, the file we want to store it in. As you can read in the help-file for this function, `append` indicates whether you want to add what you are exporting to the bottom of an already existing file. If this is set to `FALSE`, files that already exist will simply be overwritten. To reproduce the file we imported, we would need to set:

```
write.table(drm0, './data/export.csv',
  quote = FALSE, sep = ',',
  row.names = FALSE)
```

This results in a file that looks like this:

```
## text,is_lure,key,yes_key,rt,subject
## Bier,neutral,l,a,2.18085813522,0
## Dinosaurier,neutral,l,a,0.981130123138,0
## Volkswagen,correct,a,a,1.03971600533,0
## Sibirien,neutral,l,a,2.89859485626,0
```

Closing words

In this chapter we took a "quick" glance at the basics of R. Starting to get along with R can feel pretty overwhelming at first, but always remember that there is no penalty for trying something and getting it wrong a couple of times. When seeing warnings and errors, don't panic. Just remember to use `help` frequently.

If you followed along with all the examples in this chapter you should have a general idea of how to use R as a calculator (*Commenting and basic functionality*) and understand the basic rules of how to use functions (*Functions and arguments*). You should also know what objects are and how to create them (*Objects and the Environment*). What we will need in the upcoming sections are basic skills in *Handling data*, so we can prepare our analyses.

In the next chapter, we will dive into some more in-depth concepts of R, but if you want some other sources to broaden your knowledge, I would recommend starting with Hadley Wickhams **'Advanced R<http://adv-r.had.co.nz/>'_**. If your looking for answers to specific questions, check out **'R on stackoverflow<https://stackoverflow.com/questions/tagged/r>'_**.

References

1.4.2 Basic Visualisation

1.4.3 Descriptive Statistics

1.4.4 The Generalized Linear Model

t-test

correlation test

1.4.5 Multivariate data

1.5 Reporting the Results of Scientific Experiments

la bla

Subchapters

1.5.1 Pandoc intro - simple markup forms

1.5.2 Loop of submission, review, and revision

1.6 Advanced Methods

Subchapters

1.6.1 Data wrangling

Introduction to the tidyverse

1.6.2 Linear and generalized linear mixed-models

Subchapters

Introduction to mixed-effects modeling

Dejan Draschkow May 19, 2019

A hands-on crash course in reproducible mixed-effects modeling

This is a very preliminary resource, established for a satellite event at VSS 2019. Its aim is to establish the basis for a comprehensive book chapter. By gathering feedback as early as possible it will hopefully appeal to a broader audience.

Mixed-effects models are a powerful alternative to traditional F1/F2-mixed model/repeated-measure ANOVAs and multiple regressions. Mixed models allow simultaneous estimation of between-subject and between-stimulus variance, deal well with missing data, allow for easy inclusion of covariates and modeling of higher order polynomials. This workshop provides a focused, hands-on treatment of applying this analysis technique in an open and reproducible way. More introductory or extensive resources are currently available and I can highly recommend them.

Fixed and random effects

Linear vs. generalized linear

hello Aylin is the best <3

Contrast coding

Model comparisons

Determining the random effects structure

Facing a real data set

Power analysis for mixed models

1.6.3 Multivariate Analysis and Machine Learning

Introduction to scikit-learn

1.6.4 Special methods

bla blah

Subchapters

1.6.5 Neuroscience

1.6.6 Eye tracking

1.6.7 Online Experiments

Mechanical Turk

1.6.8 Formatting - saving data (ext.)

1.6.9 Data protection - security (ext.)

1.7 help

If you have any questions, ask . . . @gmail.com

1.8 How to contribute

If you want to contribute to the documentation, access the "docs" directory in the repository: https://github.com/VirtualDataLab/smobsc or simply click the "edit on github" button on the top of this page. In the docs directory you will find an index.rst file. This is basically where you see the main structure of the documentation with all the headers, subheaders etc. If you want to link to a new section, you have to add it under toctree. When building the documentation sphinx will go through the sections under toctree and look for .rst files with the same names in the docs directory.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Bibliography

[GV14] Pablo Garaizar and Miguel A Vadillo. Accuracy and precision of visual stimulus timing in psychopy: no timing errors in standard usage. *PloS one*, 9(11):e112033, 2014.

[MathotST12] Sebastiaan Mathôt, Daniel Schreij, and Jan Theeuwes. Opensesame: an open-source, graphical experiment builder for the social sciences. *Behavior research methods*, 44(2):314–324, 2012.

[Pei07] Jonathan W Peirce. Psychopy—psychophysics software in python. *Journal of neuroscience methods*, 162(1-2):8–13, 2007.

[Pla16] Richard R. Plant. A reminder on millisecond timing accuracy and potential replication failure in computer-based psychology experiments: an open letter. *Behavior Research Methods*, 48(1):408–411, Mar 2016. URL: https://doi.org/10.3758/s13428-015-0577-0, doi:10.3758/s13428-015-0577-0.

[VG16] Miguel A. Vadillo and Pablo Garaizar. The effect of noise-induced variance on parameter recovery from reaction times. *BMC Bioinformatics*, 17(1):147, Mar 2016. URL: https://doi.org/10.1186/s12859-016-0993-x, doi:10.1186/s12859-016-0993-x.

[WTNM13] Michael A Woodley, Jan Te Nijenhuis, and Raegan Murphy. Were the victorians cleverer than us? the decline in general intelligence estimated from a meta-analysis of the slowing of simple reaction time. *Intelligence*, 41(6):843–850, 2013.