2021 ■ Vol. 17 ■ no. 4

Python for Research in Psychology

Tutorial 3: Introduction to Functions and Libraries in Python

Matt Ross [a] ✉, Kinsey Church [a] & Damien Rolon-Mérette [a]

[a]Laboratory for Computational Neurodynamics and Cognition, School of Psychology, University of Ottawa

Abstract ■ The third introductory tutorial of our series "Python for Researchers in Psychology" aims to teach researchers about the importance of functions and libraries. First, we introduce the concept of functions. We cover the advantages of functions and how to use them with the help of basic examples, including a paired-samples (dependent-samples) t-test. Then, libraries and their included functions are discussed, including how to import them and the functionality of some of the most popular libraries for researchers in psychology. Finally, a longer, more complex example shows how functions and libraries can help accelerate your research, statistical analyses, and data visualization.

Keywords ■ Python, Psychology. **Tools** ■ Python.

✉ mross094@uottawa.ca

10.20982/tqmp.17.4.S013

Acting Editor ■ Thaddé Rolon-Mérette (University of Ottawa)

Reviewers ■ Artem Pilzak (University of Ottawa)

Introduction

When learning to code, one of the main deterrents for a novice programmer is the thought that they need to code everything from scratch. This idea that you need to code all procedures and equations by yourself can be daunting. It is a particular setback for those interested in using Python for psychology, especially when it comes to running statistical analyses. There are numerous softwares that offer a semi-automated form of data analysis and can run a number of statistical tests. However, these softwares are typically limited to running a limited number of pre-programmed statistics and can be quite expensive. Python offers a free alternative to running statistical tests and offers a limit-free environment where custom statistical tests and analyses can be performed. Using Python, all aspects of your research, not just statistics, can be coded easily by using blocks of reusable code known as functions.

Within the Python programming language, functions are used to save the user time from re-coding things over and over again. For instance, if you need to maintain anonymity by renaming all of the participants in your study each time you collect data, you could create a function that assigns them a number and stores their user-names somewhere else. This function could then be used each time you import a new dataset! Not only can functions accelerate your research, but you can also use functions that other programmers have created and import libraries full of useful functions for performing specific tasks.

Libraries in Python are essentially groups of useful functions that are already written. This means that once you import a library, you have access to these prebuilt functions. For example, if you need to create a function for processing electroencephalography (EEG) and magnetoencephalography (MEG) data, you could look for one that has already been created by another researcher and find MNE, a library that is already built, and just simply import it into your script. This is another advantage of Python being an open source language: you can use code from others with ease.

This tutorial aims to get you comfortable with functions and libraries so that you can take advantage of these time savers in your own coding projects. First, it will cover the basics of functions, how to use them, and how to build your own from scratch, including several examples. Next, it will cover the advantages of libraries, how to import and use them, and some of the key libraries for researchers in psy-

Figure 1 ■ A basic function called `display()` that prints a given input.

```
In [1]: def display(input):
            print(input)

In [2]: display("Psychology Rules")
        Psychology Rules
```

chology. Finally, a more detailed example will demonstrate how to use different functions from different libraries on the same dataset in order to perform statistical analyses and data visualization.

Functions

As previously mentioned, functions are reusable blocks of code. By reusing your code, you can save time from not re-coding things and make your script more efficient. When using functions, your script will have fewer lines of code because the same commands do not have to be rewritten over and over (Matthes, 2019). Also, a unique characteristic of functions is that they do not save all of the variables used within them. This is particularly beneficial when you have large datasets, as less information is saved to your computer's memory. When these factors are combined, you save a lot of time and effort and your code is more efficient!

Functions are easy to create and apply to a wide range of problems. Once created, there is no limit on the amount of times that you can apply your function to different problems. We are going to show you how to create your own by starting with a simple function that displays any text the user wants. To build a function, there are a few key steps (Van Rossum and Drake Jr, 1995). First, you need to name the function using the keyword `def`, which is an abbreviation for the word "define."

For this example (see Fig. 1), we are going to name/define our function "display", since it gives a short, clear understanding of what the function does. It is good practice to name your functions something clear, concise, and descriptive so you will know exactly what they do later on!

After defining the name of your function, you will need parentheses to define your arguments, or inputs. These arguments are the variables that will be used in the function. In this case, we only need one input (named *input*) which will be what we want displayed. Next, you will need a colon to start the function and all of the lines inside of it must be indented. The first indented line of our function will be to display our input using the `print()` function. Finally, to end the function, simply stop indenting your code or use a `return()` function.

To execute the function, the name needs to be called with the variables that will be used as input. In this case, we call our newly created `display()` function and using parentheses we enter the variable that we wish to be displayed. The input here is a string that reads "Psychology Rules". Once the line of code that calls our function is executed, the phrase "Psychology Rules" will be displayed. This function can be re-used to display any phrase that the user would like (Matthes, 2019).

Example of creating a function with a paired-samples t-test

Now that we have covered the basics of the structure of functions, we can extend this to the commonly used paired-samples t-test in psychology. A paired-samples t-test calculates whether the means of two measurements taken from the same participants are statistically different or not. In this example, a repeated-measures design is used where the same participant reports their measurements before and after treatment (Privitera, 2017).

Example 2 shows a fake dataset of measures of anxiety on test-taking pre-treatment and post-treatment, data points on a fictional Likert scale ranging from 1-10 (see Fig. 2). The goal is to see if mindfulness meditation (the treatment) improves participant's perceived anxiety for test-taking. Mindfulness meditation has been described as: "sitting quietly and is mainly characterized by just observing one's experiences, not creating or modifying them" (Khoury et al., 2013). In order to measure the effect that mindfulness meditation has on test-taking anxiety, we need to conduct a paired-samples t-test (Field, 2016) in Python using a function.

Before we can write our t-test function, we need to count the number of participants in each group and set the sample size. This is done in a few quick lines of code (see Fig. 2). Using the pre-treatment and post-treatment data, we can apply the `len()` function to each group to get the number of participants, our sample size. A quick check of equal sample sizes is done between groups to ensure they are the same size. Since both groups have an equal sample size, we can assign the value of 10 to our sample size variable n.

We are going to begin our analysis process by creating a function that will calculate the mean of our sam-

Figure 2 ■ Dataset for Example 2 and the steps needed before writing a t-test function.

```
In [3]: pre_treatment = [7,9,4,10,3,8,9,5,5,7]
        post_treatment = [5,6,2,4,3,6,6,6,5,3]
        alpha = 0.05

In [5]: n_pre_treatment = len(pre_treatment)                    # count the number of participants in the pre-treatment sample
        n_post_treatment = len(post_treatment)                  # count the number of participants in the post-treatment sample
        print("pre-treatment sample size = ", n_pre_treatment)  # display the result
        print("post-treatment sample size = ", n_post_treatment) # display the result

        pre-treatment sample size =  10
        post-treatment sample size =  10

In [6]: n = 10                                                  # set sample size (n) to 10
        print("sample size = ", n)                              # display the result

        sample size =  10
```

Figure 3 ■ Function called `mean_function()` for calculating the mean of our sample groups.

```
In [7]: #function for calculating mean
        def mean_function(data,sample_size):
            calculated_mean = sum(data)/sample_size          # calculate the mean
            return(calculated_mean)

In [8]: mean_pre_treatment = mean_function(pre_treatment,n)  # apply the calculate mean function for anxiety pre-treatment
        print("mean pre-treatment = ", mean_pre_treatment)   # display the result

        mean pre-treatment =  6.7

In [10]: mean_post_treatment = mean_function(post_treatment,n) # apply the calculate mean function for anxiety post-treatment
         print("mean post-treatment = ", mean_post_treatment)  # display the result

         mean post-treatment =  4.6
```

ples. To begin, we will use `def` to define our function as "mean_function" (see Fig. 3). Next, we will use our parentheses to tell our mean function that it will be using the sample data (*data*) and the sample size (*sample_size*). This means we have two inputs that are required in order for our function to run. You can add as many arguments to a function as you need! After defining the variables that are used by the function, we always follow with a colon and indent the next lines.

To calculate the mean, we take the sum of our dataset and then divide it by the given sample size (see Equation 1). Our function is ended by using the `return()` function to return the newly calculated mean (see Fig. 3; for more technical descriptions, please see Matthes, 2019). Once defined, we can call our function using the data from each group (*pre-treatment* and *post-treatment*) and our sample size (n) to get the means of our pre-treatment and post-treatment groups.

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{1}$$

Now that we can calculate the mean and we have the sample size of our datasets, it is time to write the t-test function (see Fig. 4). It is important to note that functions can be nested within each other, so we are able to include the `mean_function()` from above into our final t-test function. We begin by defining the function as "calculate_t_test" and then adding our three arguments: one for the pre-treatment data called *pre_data*, one for the post-treatment data called *post_data*, and one for the size of both samples called *sample_size*.

The t-test function begins by calculating the means of both datasets using the `mean_function()`. Then, two empty lists are created to store the calculated differences and squared differences between the pre-treatment and post-treatment datasets (*differences* and *squared_differences*). Next, a `for` loop is used to calculate the differences and squared differences between the *pre_data* and *post_data* for each participant (for the total length of the sample_size). The sum of differences and squared differences are then calculated using the `sum()` function and stored in their respective lists.

Next, the standard deviation and standard error can be calculated (see Equations 2 and 3) by using the sum of *squared_differences*, the sum of *differences*, and the *sample_size*. Finally, the t-statistic is calculated (see Equation

Figure 4 ■ Final t-test function `calculate_t_test()` and results.

```
In [11]:
#function for calculating mean
def mean_function(data,sample_size):
    calculated_mean = sum(data)/sample_size                # calculate the mean
    return(calculated_mean)

In [12]:
#create t-test function
def calculate_t_test(pre_data,post_data,sample_size):
    mean_pre_treatment = mean_function(pre_data,sample_size)  # apply the calculate mean function for anxiety pre-treatment
    mean_post_treatment = mean_function(post_data,sample_size)# apply the calculate mean function for anxiety post-treatment
    # create an empty list to store differences and squared differences for each participant pre and post-treatment
    differences = []
    squared_differences = []
    # use a for loop to iterate over each participant to calculate pre and post-treatment
    for i in range(sample_size):
        differences.append(pre_data[i]-post_data[i])          # calculate difference and store in the list
        squared_differences.append((pre_data[i]-post_data[i])**2) # calculate square difference and store in the list
    sum_differences = sum(differences)                        # get the sum of all participant differences
    sum_squared_differences = sum(squared_differences)        # get the sum of all participant squared differences
    # calculate standard deviation
    standard_deviation = ((sum_squared_differences-(sum_differences**2/sample_size))/(sample_size-1))**(1/2)
    # calculate standard error
    standard_error = (standard_deviation/sample_size**(1/2))
    # calculate observed t statistic
    observed_t_statistic = (mean_pre_treatment-mean_post_treatment)/standard_error
    # calculate the degrees of freedom
    degrees_of_freedom = sample_size-1
    return(observed_t_statistic,degrees_of_freedom)

In [13]:
t_statistic,df = calculate_t_test(pre_treatment,post_treatment,n)
print("Observed t value = ", t_statistic)
print("Degrees of freedom = ", df)

Observed t value =  3.194226797349382
Degrees of freedom =  9
```

4) by using the calculated means from above and the standard error from above. The degrees of freedom (*df*) are also calculated from the given sample size so it is easier for the user to analyze their results. To end the function, the `return()` function is used, returning the *observed_t_statistic* variable and the *degrees_of_freedom* variable. It is important to note that you can return any variables used in a function, by simply adding them to the return command.

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n}(x_{pre_i} - x_{post_i})^2 - \frac{\sum_{i=1}^{n}(x_{pre_i} - x_{post_i})}{n}}{n-1}} \quad (2)$$

$$SE_{diff} = \frac{\sigma}{\sqrt{n}} \quad (3)$$

$$t_{obs} = \frac{\bar{x}_{pre} - \bar{x}_{post}}{SE_{diff}} \quad (4)$$

To execute this function, `calculate_t_test()` is used with our *pre_treatment*, *post_treatment*, and n variables from above being used as input. This returns our *observed_t_statistic* variable and the *degrees_of_freedom* variable, which are then assigned to the new variables *t_statistic* and *df*. It is important to note: when a function returns a variable, you must create variables that store this information outside of the function. To do this, we create as many variables as are returned on the left side of the equals sign. Afterwards, we can `print()` out our two results. In this case, we get an observed t-statistic of 3.194 and the degrees of freedom equal 9. By looking up a one-tailed t-table and using an alpha level of 0.05 (Privitera, 2017), we find that our t-critical is 1.833 and our results are significant ($p < 0.05$)! In this fake data, the mindfulness meditation had a significant effect on reducing test-taking anxiety.

The power of functions is that once they are written, they can be applied time and time again. This is particularly beneficial to psychology researchers, as we often run the same test many times or run multiple studies to replicate our results. For the next example, we wanted to replicate our results from our first experiment. To achieve this, we generated a second sample of fake participant data. We can quickly apply our `calculate_t_test()` function a second time to get the t-statistic for our new study (see Fig. 5). In this case, our observed t-statistic value was 0.128 with a total of 9 degrees of freedom. Since our t-critical value of 1.833 has not changed, we find that the result of mindfulness meditation had no effect ($p > 0.05$) on the level of test-taking anxiety for this sample.

As you can see from our examples, building your own functions can be very useful and you can customize them to do whatever you want. However, sometimes building a function takes a lot of effort and the perfect function already exists for what you are trying to accomplish. All you

Figure 5 ■ Applying `calculate_t_test()` function to another dataset and results.

```
In [14]: second_pre_treatment = [5,2,6,7,3,8,1,5,5,3]
         second_post_treatment = [5,6,2,4,3,6,4,6,5,3]
         second_n = 10                                  # set sample size (n) to 10

         second_t_statistic,second_df = calculate_t_test(second_pre_treatment,second_post_treatment,second_n)
         print("Observed t value = ", second_t_statistic)
         print("Degrees of freedom = ", second_df)

         Observed t value =  0.12803687993289553
         Degrees of freedom =  9
```

Figure 6 ■ How to import the stats module from the Scipy library.

```
In [1]: from scipy import stats
```

need to do is import the correct library to access it.

Libraries

As mentioned above, libraries are a collection of prebuilt functions that can be easily called into a script. These libraries are quick to install and there are already many existing libraries in Python, meaning there are all kinds of functions for any test or procedure you can imagine. For examples of some of the most popular libraries and what they are used for, see "Key Libraries" below.

Let us look at how libraries and their repository of prebuilt function can be used to accelerate your programming desires. To begin with, think back to the t-test example in this tutorial, where a fictional dataset contained two lists of pre-treatment and post-treatment scores. In that section, the goal was to demonstrate how to build functions and why they are considered very useful. In a few lines of code, it was possible to create a paired-samples t-test that would take the two dependent lists of data and determine if they were significantly different from one another. Here, the goal is to demonstrate how Python's community can be used to simplify this process. There are various libraries that contain prebuilt functions of every type. One such library is Scipy, where a wide range of statistical functions can be found. Let us take a look at how it could have been used on the first example's dataset to perform the same paired-samples t-test.

The first step is to import the stats module from the Scipy library (see Fig. 6). To import, you can simply type "import scipy" or if you only need one section of the library, you can import that section specifically. In this case, we just need the stats module of the Scikit_Learn library, so we can type "from scipy import stats". When possible, it is important to import only the commands that will be used. Doing so ultimately saves time and memory and is considered proper coding practice.

The second step is to regenerate the pre-treatment and post-treatment lists of data (see Fig. 7).

The final step is to use the stats module to have access to the prebuilt `ttest_rel()` function. This is done by calling the module name followed by a dot and then the function name. It can then be used on the dataset (see Fig. 8).

Look how amazing and simple that was! In four lines of code, a paired-samples t-test was executed and the results are identical to the first example ($p < 0.05$). In addition, the prebuilt function gives us the exact p-value, meaning we no longer have to estimate based on a t-table. This is why it can be advantageous to look for pre-existing functions in libraries before coding your own from scratch.

Key Libraries

- Numpy is a Python library that is used for working with arrays (collections of items). It is good for simple math

Figure 7 ■ Enter the same datasets as Example 2.

```
In [2]: pre_treatment = [7,9,4,10,3,8,9,5,5,7]
        post_treatment = [5,6,2,4,3,6,6,6,5,3]
```

Figure 8 ■ Using the prebuilt `ttest_rel()` function from Scipy to perform the same t-test as Example 2.

```
In [3]: stats.ttest_rel(pre_treatment,post_treatment)
Out[3]: Ttest_relResult(statistic=3.194226797349381, pvalue=0.010932133015728833)
```

Figure 9 ■ Importing all of the required libraries.

```
In [1]: import pandas as pd
        import numpy as np
        import seaborn as sb
        import matplotlib.pyplot as plt
        from scipy import stats
```

and built-in functions, such as `np.append()`, which adds items to the end of an array. It can be used to create vectors and matrices, which can be useful when storing large sets of participant data. Also, it can be used to generate random numbers from specified various distributions, for instance a normal distribution, with `np.random.normal()`.

- `Pandas` is a Python library that is used for data analysis. It allows you to create or load existing datasets and manipulate them with various operations. It easily transitions between .csv files, Excel files, and Python!
- `Matplotlib` is excellent for simple plots and allows you to control your colours, legends, marker type and size, and many other parameters. It can also be used to plot more complex graphics such as decision boundaries.
- `Seaborn` is a library based on Matplotlib that can help you create ready-to-publish graphics.
- `Scipy` is a library for scientific computations. For instance, this library contains functions for statistics, linear algebra, and signal processing.

Complex Example

Now we will tackle a more complete example by downloading a dataset from the Kaggle website, visualizing it through the pandas library, extracting the columns that are of interest, pre-processing them, performing a paired-samples t-test, and visualizing the data's distribution using Seaborn.

To start, we downloaded the appropriate dataset from the Kaggle website. It can be found under "College Basketball Dataset" and we will download the entirety of the available data (Kaggle basketball dataset). From there, we will ensure that the downloaded file is located in the same folder as the Python script that will be used to analyze it. Once everything is in the same location, we will import the Pandas library to allow us to manipulate the dataset. Another good coding practice is to shorten the name of your libraries when you import them. For instance, using "import pandas as pd" makes it so that we can access Pandas library by typing "pd" instead of typing out "pandas." To save us some time, we will also import all the libraries that we will use in this section (see Fig. 9).

The next step will be to load the datasets and save them as variables. Our goal is to determine if all teams perform consistently from one year to the next or if any of them differ. Thus, we will compare the adjusted offensive efficiencies *ADJOE* of each team in 2014 and see if they are significantly different in 2015. Figure 10 shows how the `pd.read_csv()` function can be used to store the entirety of the 2014 ("cbb14.csv") and 2015 ("cbb15.csv") data into dataframes (*df_2014* and *df_2015*).

From here, it is important to see the dataframes to better understand the data you are working with. This can be done by using the `pd.head()` function, which shows the first five rows of all of the columns (see Fig. 11). As you can see, there are various columns of variables that can be used to perform our analysis, but we will focus on the *ADJOE* column (adjusted offensive efficiency). This process should also be repeated for 2015 to ensure that the datasets

Figure 10 ■ Import the datasets for 2014 and 2015 into a dataframe using the `pd.read_csv()` function from Pandas.

```
In [4]: df_2014_proc=df_2014.filter(items=['TEAM','ADJOE','W']).sort_values('TEAM',ascending = True).reset_index(drop=True)
        df_2015_proc=df_2015.filter(items=['TEAM','ADJOE','W']).sort_values('TEAM',ascending = True).reset_index(drop=True)
```

Figure 11 ■ Using the `pd.head()` function from Pandas to visualize the start of the dataframe.

```
In [3]: df_2014.head()
```

Out[3]:

	TEAM	CONF	G	W	ADJOE	ADJDE	BARTHAG	EFG_O	EFG_D	TOR	...	FTR	FTRD	2P_O	2P_D	3P_O	3P_D	ADJ_T	WAB	POSTSEASON
0	Arizona	P12	38	33	116.2	87.4	0.9636	51.7	42.3	15.7	...	41.0	34.2	50.7	40.2	36.4	32.0	64.3	9.4	E8
1	Florida	SEC	39	36	115.9	88.4	0.9575	52.2	45.4	17.5	...	42.4	31.2	51.3	43.5	35.9	33.0	63.1	11.7	F4
2	Virginia	ACC	37	30	114.6	89.5	0.9449	50.8	44.2	16.5	...	42.0	32.5	49.0	42.1	36.9	32.3	61.2	8.2	S16
3	Wichita St.	MVC	35	34	116.4	93.0	0.9295	52.4	44.7	16.2	...	46.9	35.7	52.3	43.6	35.0	31.1	65.1	8.1	R32
4	Kansas	B12	35	25	119.2	95.6	0.9270	54.2	47.1	19.1	...	48.3	45.5	55.5	44.3	34.0	35.5	68.2	6.9	R32

5 rows × 23 columns

Figure 12 ■ Filtering the datasets for the 2014 and 2015 season data for pre-processing.

```
In [2]: df_2014=pd.read_csv('cbb14.csv')
        df_2015=pd.read_csv('cbb15.csv')
```

are equivalent in terms of size (same number of rows and columns) and content.

The next step is to isolate the *TEAM*, *ADJOE* and *W* columns to reorder them so we can adequately compare the same team from the 2014 season to the 2015 season. This is done by using the `filter()`, `sort_values()`, and `reset_index()` functions from Pandas. Once we pre-process this data we will save it as a new dataframe to ensure we do not lose our original dataset (see Fig. 12).

From there we can let the magic begin! From the Numpy (imported as "np") library we can extract the general statistics of our two datasets, such as the mean and the standard deviation. This is done by using the `np.mean()` the and `np.std()` functions from Numpy. Furthermore, we can use the stats module's `ttest_rel()` function (previously described) to perform our paired-samples t-test (see Fig. 13 for how this is implemented).

Finally, we can use the Seaborn library (imported as "sb") to visualise the distribution of our two datasets. With the `sb.displot()` function, we can plot a histogram of our two datasets together (see Fig. 14).

The results are astounding. Instead of having to focus all of our energy in developing each function, we can use what has been created by others in the Python community for free to help us in our analyses. That is one of the reasons why programming in Python is so popular and efficient! We hope that this tutorial will inspire you to start using functions and libraries in your own programming and research.

Authors' note

This work was supported by the Ontario Graduates Scholarship (OGS) and the Fonds de recherche du Québec - Nature et technologies (FRQNT) award programs.

References

Field, A. (2016). *An adventure in statistics: The reality enigma*. Sage publications.

Khoury, B., Lecomte, T., Fortin, G., Masse, M., Therien, P., Bouchard, V., ... Hofmann, S. G. (2013). Mindfulness-based therapy: A comprehensive meta-analysis. *Clinical psychology review*, *33*(6), 763–771.

Matthes, E. (2019). *Python crash course: A hands-on, project-based introduction to programming*. no starch press.

Privitera, G. J. (2017). *Essential statistics for the behavioral sciences*. Sage publications.

Van Rossum, G., & Drake Jr, F. (1995). Python tutorial (vol. 620). *CWI Report CS-R9526, Amsterdam, Netherlands, msekce. karlin. mff. cuni. cz/˜ halas/IT/tutorial. pdf*.

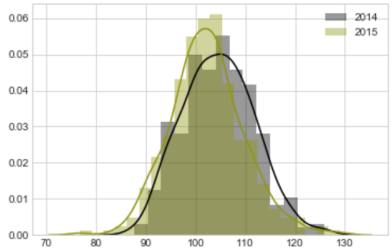
Figure 13 ■ Using the `np.mean()` and `np.std()` functions on the dataframes.

```
In [5]: print('2014 Mean:', np.mean(df_2014_proc['ADJOE']),'2014 STD:',np.std(df_2014_proc['ADJOE']))
        print('2015 Mean:', np.mean(df_2015_proc['ADJOE']),'2015 STD:',np.std(df_2015_proc['ADJOE']))
        print(stats.ttest_rel(df_2014_proc['ADJOE'],df_2015_proc['ADJOE']))

        2014 Mean: 104.5849002849003 2014 STD: 7.1315897644497515
        2015 Mean: 102.31538461538463 2015 STD: 7.390954069675187
        Ttest_relResult(statistic=8.29784432838373, pvalue=2.3254224023939524e-15)
```

Figure 14 ■ Using the `sb.displot()` function to display our results in a beautiful plot.

```
In [6]: plt.style.use('seaborn-whitegrid')
        sb.distplot(df_2014_proc["ADJOE"],kde=True,color='k')
        sb.distplot(df_2015_proc["ADJOE"],kde=True,color='#8f9805')
        plt.legend(labels=['2014', '2015'])
        plt.xlabel('')
        plt.show()
```



Citation

Ross, M., Church, K., & Rolon-Mérette, D. (2021). Tutorial 3: Introduction to functions and libraries in python. *The Quantitative Methods for Psychology*, *17*(4), S13–S20. doi:10.20982/tqmp.17.4.S013

Received: 04/09/2021 ~ Accepted: 04/11/2021