

## Python for Research in Psychology

# Introduction to Python's Syntax

Kinsey Church<sup>a</sup> , Thaddé Rolon-Mérette<sup>a</sup> , Matt Ross<sup>a</sup>  & Damien Rolon-Mérette<sup>a</sup> 

<sup>a</sup>Université d'Ottawa

**Abstract** ■ This tutorial is the second in our series and covers basic syntax in Python with examples related to psychology. The aim is to teach programming beginners and experts alike the fundamentals required in order to smooth the learning curve and succeed with integrating Python with their research. It starts by covering basic built-in functions and variable creation. Next, different data types and data structures that you will encounter are covered detail, followed by comments and best commenting practices. Finally, indentation, logic, conditional statements, and loops are all explained with simple, illustrative examples. The tutorial ends with a comprehensive example of the same-different task from cognition that ties together everything learned. With this foundation, the reader will gain the confidence to begin practicing Python on their own and think of ways to incorporate it into their own research and daily lives.

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

✉ [KCHUR026@uottawa.ca](mailto:KCHUR026@uottawa.ca)

 [10.20982/tqmp.17.1.S001](https://doi.org/10.20982/tqmp.17.1.S001)

**Acting Editor** ■  
[Nareg Berberian](#)  
(University of Ottawa)

**Reviewers**  
■ [Matias Calderini](#)

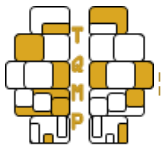
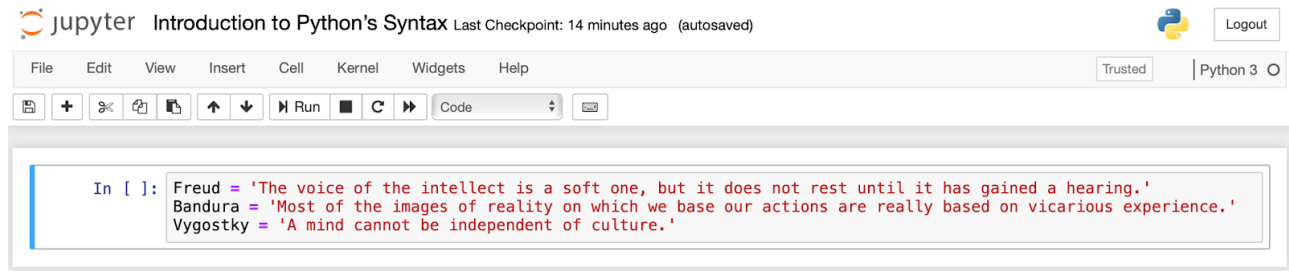
### Introduction

Learning Python is like learning a new language. To master it, it is important to have a strong foundation in its structure and different components, much like understanding the sentence structure and parts of speech of any language. While this may seem challenging, there are a number of resources that can be used to soften the learning curve (Harwani, 2012; Ngo, 2017; Pine, 2019; Summerfield, 2010; Van Rossum and Drake, 2011). Unfortunately, the majority of them are not tailored towards the field of psychology. This tutorial aims to fill in this gap by covering the basics of Python's syntax and introducing examples relevant for psychology. It is highly recommended to follow the previous tutorial first: "Introduction to Anaconda and Python: Installation and Setup." To better learn from the examples presented in this tutorial, we recommend that you open Jupyter notebook to follow along with the examples presented. The advantage of using an integrated development environment (IDE), such as Jupyter notebook, is that it will colour code the different aspects of your script to make it easier to read and provide debugging assistance. This tutorial is divided as follows: an introduction to variables, data types, data structures, comments, indentation, logic state-

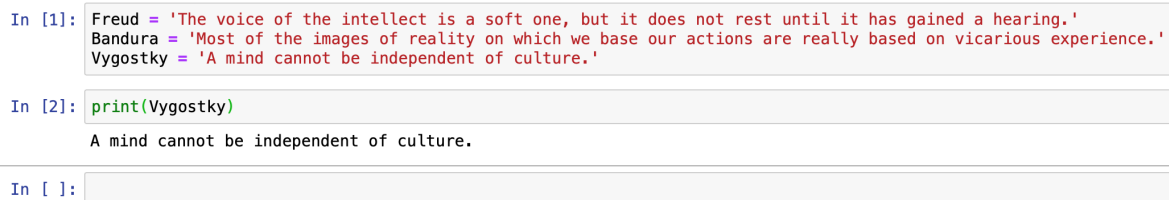
ments, loops, and a comprehensive example related to the field of psychology.

### Getting Started

We begin the new project by opening the Jupyter notebook IDE. As per any new project, we will start by creating a new notebook/Python script, naming it, and saving in a desired location. In this project, we are creating three variables, each named after a famous psychologist (Freud, Bandura and Vygotsky) and will attribute a quote to each one. To accomplish this, we use a single equal sign (=) to specify that the variable name (Freud) contains/represents what is on the other side of the equal sign ("The voice of the intellect is a soft one, but it does not rest until it has gained a hearing.'). It is important to put quotation marks around the attributed phrases. This will change the colour of the quote from black to red and create what is called a string (more on this later). Once all three variables are correctly defined as per Fig. 1., press 'Run' to execute the cell. This will save your variables and their contents for future use. A quick example would be to use the **print()** function to display each quote. To use this built-in function, simply write **print()** and insert your variable name in-between the parentheses. Once completed, execute your coding cell

**Figure 1** ■ Creating variables.

```
In [ ]: Freud = 'The voice of the intellect is a soft one, but it does not rest until it has gained a hearing.'  
Bandura = 'Most of the images of reality on which we base our actions are really based on vicarious experience.'  
Vygostky = 'A mind cannot be independent of culture.'
```

**Figure 2** ■ Example of using variables in a function.

```
In [1]: Freud = 'The voice of the intellect is a soft one, but it does not rest until it has gained a hearing.'  
Bandura = 'Most of the images of reality on which we base our actions are really based on vicarious experience.'  
Vygostky = 'A mind cannot be independent of culture.'  
  
In [2]: print(Vygostky)  
A mind cannot be independent of culture.  
  
In [ ]:
```

and the results should be identical to Fig. 2. Congratulations, you have created your first few variables!

**Note:** You may have noticed that different colours have appeared in your notebook. These colours are used to help differentiate between data types. Just as nouns, verbs and adjectives are all different and have special uses in a phrase, so are the variables, strings and functions (just to name a few) in a script. Table 1 sums up the meaning of each default colour for three popular IDEs.

Creating variables is very important for saving and reusing information. Think of creating an experiment where in each trial you must display the same prompt to your participants. Instead of having to copy/paste your entire prompt for each trial, you can save it as a variable and use it repeatedly. This allows you to save time and avoid lengthy code. In Python, you can attribute a variety of information to variables, such as numbers, phrases and even entire functions. That being said, there are a few important tips to remember when creating a variable:

- Spaces are not allowed in variable names, but underscores are commonly used to separate words. For example, *psychology\_rocks* works, but *psychology rocks* will cause an error.
- Variable names can contain only letters, numbers, and underscores. Most importantly, they cannot start with a number. For instance, you can call a variable *Bandura\_1* but not *1\_Bandura*.

- Avoid using Python’s predefined function names as variable names, such as the word “print”.
- Variable names should be short but descriptive. For example, *name* is better than *n* and *student\_name* is better than *s\_n*. Once you start building very long scripts, you may start to lose track of what some of your variables mean. Good coding practice includes naming your variables something meaningful to improve readability for yourself and others.

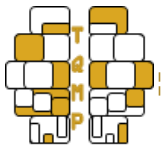
Now that you have a better understanding of what variables are, let’s dive into the specific data types that can be saved and stored.

### Data Types

There are several different data types that are available in Python, each with its own utility. Outlined below are the four most common data types you will encounter. For each one, a quick explanation and a simple example of how it may pertain to research in psychology can be found. Let’s begin learning the ABCs of Python!

#### Strings

A string is a sequence of characters, anything that you type (a letter, a number, a symbol, or a space), enclosed in ‘single’ or “double” quotation marks. An example of how strings can be used is when a researcher records a subject’s response to the open ended question “How has your



depression been affecting your life?” (see Fig. 3.). Strings are immutable, which means that you cannot change the contents of a string after it has been created, you can only write over it (redefine the variable).

### Integers

Integers are positive or negative whole numbers (no decimal point). An example of this could be how a researcher in the field of neuroscience would count the number of active neurons at a given time (see Fig. 4.).

### Floats

Floats, or float point real values, are real numbers (including numbers with a decimal point). An example of a float would be recording a participant’s reaction time in milliseconds to a given task (see Fig. 5.).

**Note:** In certain cases, you may encounter floats or integers as either 32 or 64 bit. This simply refers to the number of bits you are limiting the number to be. Essentially, float64 has double the number of values after the decimal point and requires more bits than float32.

### Booleans

Booleans are also known as logical expressions, meaning that they are evaluated as **True** or **False** (1 or 0). An example would be responses to a questionnaire whereby a participant indicates positively or negatively to experiencing anxiety or depression (see Fig. 6.).

At any time, the type of data stored in a variable can be easily checked using the built-in **type()** function. The output from this function will indicate the data type (see Fig. 7.).

### Data Structures

If data types are the ABCs of Python, then data structures would be words. They define how your variables are structured and show how to store multiple data types in the same variable. Outlined below are four data structures that you will encounter while coding.

#### Set

A set is a group of objects that is unordered, where no duplicates are allowed. Delimited by { }, sets can be used to store objects of any data type (string, integers, etc.). An example would be creating a set for storing the symptoms of a particular disorder, such as ADHD (American Psychiatric Association, 2000), where the order of symptoms is not important (see Fig. 8.).

#### List

Lists are ordered sequences of objects that are delimited by [ ]. Lists may contain objects of any data type, and data

types can even be mixed within the same list. An example of a list would be for storing IQ scores for a group of participants, where the position of the score in the list (index) matches the participant’s identification number (See Fig.9.).

Since lists are ordered, you can use indexing to retrieve information from a specific position in a list by using square brackets after the variable name. Indexing in the Python language starts at 0, meaning the first object in a list is 0, the second is 1, etc. For example, if a researcher is interested in the first participant’s IQ score, they could use `IQ_scores[0]` to retrieve the first object in that list (see Fig. 10.).

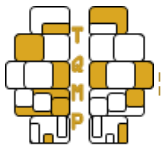
#### Tuple

Tuples are similar to lists, as they are ordered sequences of objects. However, they are immutable (cannot be altered, only overwritten) and are delimited by ( ). A benefit of creating an immutable object is that they take up less space in the computer system’s memory. An example of a tuple would be storing an individual participant’s reaction time and whether they are part of the control group or not (see Fig. 11.).

#### Dictionary

Dictionaries are similar to sets as they are unordered, but they are a little different. They have a unique format that allows them to be accessed in a similar way to indexing. Dictionaries use objects known as “keys” to retrieve other stored objects, known as “values”. In a dictionary, these “keys” cannot be repeated (duplicates), but the “values” stored inside can be repeated as much as needed. In addition, the objects are mutable, meaning that your code can update and change these values when specified to do so. An example where a researcher might use a dictionary is for storing information about different species of rodents, like whether a particular species of rodent has a specific gene or not and their average weight in grams. To achieve this, the researcher would create an empty dictionary for `Rodent_species` using {}. They could then create a “key” for each species of rodent and create a map that indicates various characteristics (or “values”) that are specific to that species, such as the presence of a gene and the average weight (see Fig. 12.).

To access specific information in a dictionary, similar to indexing, square brackets are used. However, instead of a numeric value indicating position, the “key” is used, as dictionaries are unordered. Relating back to the previous example, if a researcher had a dictionary of rodent species with relevant information stored, but was only interested in the *Mus musculus* species, they could specify this “key” and retrieve all the mapped “values” related to this species

**Figure 3** ■ String example.

```
In [1]: subject_response = "I am having trouble sleeping"
```

**Figure 4** ■ Integer example.

```
In [4]: Number_active_neurons = 472
```

(see Fig. 13.).

**Note:** For a summary of all the different data structures, see **Table 2** in the appendix.

### Comments in Python

In order to stay organized, Python's programming language provide users with an easy way to label or annotate sections of their code. Referred to as a comment, these can be used to remind yourself what a certain section of code is doing and label this information. This is specifically useful if you are working with others. Python will completely ignore your comments, meaning that they will not change the way the code is run or interpreted. This means you can use comments to isolate certain lines of code temporarily and verify if it is working as intended. The main way to comment in Python is to use the hash mark (#). Common practice is to use these comments on a separate line or inline with other code. See Figure 14 for an example of how comments can be used to clarify some of the examples given above.

**Note:** It is a very important habit to always write comments in your code. As time progresses, your skill will change and you will forget what certain variable's acronyms mean as well as what some functions are doing. Commenting will save you time in the long run and will avoid future confusion. Also, commenting is very useful for others that are trying to use or replicate your code.

### Indentation

A block of code is related lines of code that are grouped together for a specific purpose. For example, if several dif-

ferent sections are all part of the same function, they would be found in the same block. In order to define these blocks, many languages use curly brackets {}, but Python uses indentation. This means that each line pertaining to a block is indented by the same amount to the right. It is also possible to indent multiple times, effectively combining multiple blocks of code, embedding one within another. This is known as nesting and helps the software understand which sections of your code take priority. Fun fact: This is where Python gets its name! Once your script is large and involves several blocks of code, it will start to look like a snake slithering due to the indentation. You will see this effect of indentation for the rest of this tutorial.

### Logic in Python

As per any other programming language, Python has various ways of activating certain blocks of codes (conditional, if, elif and else statements). For instance, if you have a block of code that should only activate if a participant's name is provided, you can do so by using conditional and logical statements. These are special types of code that allow any user to incorporate logic into their program. Think of mathematical conditions, such as the symbols <, >, if, etc., these can all be used to determine which block of code should be executed. When checking if the coding conditions are met, booleans are employed that either return a **True** or **False** value. This determines if the block of code should be executed (**True**) or not (**False**). To make this more concrete, let's dive into a few examples to see how they work.

**Figure 5** ■ Float example.

```
In [4]: Reaction_time = 1.25674
```

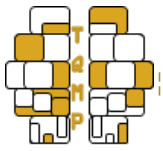


Figure 6 ■ Boolean example.

```
In [1]: experience_anxiety = True
        experience_depression = False
```

Figure 7 ■ Example of using type().

```
In [2]: experience_anxiety = True
        experience_depression = False

        type(experience_anxiety)

Out[2]: bool
```

### Conditional Statements

As the name implies, it is a statement that is either **True** or **False** based on a particular rule/condition. You have already encountered many of them through mathematics. Examples of common conditional statements that can be found in Python are the greater than `>`, smaller or equal to `=<` and not equal to `!=`, just to name a few. To be more precise, a conditional statement is a way for a programmer to create a logical rule that can be incorporated into a script. For a full description of the most common conditional statements, their meaning, and a simple example, see Fig. 15.

**Note:** the `==` and `!=` statements can also be used for other data types as well (such as strings). This can be very helpful, especially to verify if duplicates exist in a database. Furthermore, there is a fundamental distinction to remember between the `=` and `==` signs. The first is always used to assign a variable to an object (as seen at the beginning of the tutorial) and the second is used to verify if two objects are the same (is the string “Hello” the same as “Bonjour”).

There are also ways of combining two or more conditional statements by using logical operators. The main three are the **and**, **or**, and **not** operators. The **and** combines conditional statements and requires all of them to be met to return a **True** value. The **or** allows more flexibility by returning a **True** value as long as at least one of the conditional statements are met. The **not** verifies if a con-

ditional statement is not met and if this is the case, it will return a **True** value. Fig. 16. illustrates a small example for each operator.

### If, elif and else Statements

You can also use conditional statements to either execute (run the code) or to simply bypass it (ignore it). These **if**, **elif** and **else** statements rely on the value of the booleans to determine their action. In other words, if the condition is met, execute the desired block of code, if not, ignore it. The **if** statement implies if **True**, run the block of code. The **elif** is a secondary **if** statement and stands for “else if”. When the first **if** statement’s conditions are not met, move on to the **elif** statements and check their conditions. You can use as many **elif** statements in a row as needed. The **else** statement is executed if none of the previous **if** and **elif** statements are **True**, and encompasses all remaining possible conditions. In other words, the logic is as follows: Run the block of code above if any of the **if** or **elif** statements are **True**. If none of these statements are met, run the **else** block of code. A simple example to better understand this would be to create an **if**, **elif** and **else** statement that prints out a participant’s answer as either being correct, incorrect or invalid based on their values (see Fig. 17.).

To solidify this concept, let’s re-examine this code, but this time in simple phrase format:

Figure 8 ■ Set example.

```
In [13]: ADHD_symptoms = {"Inattention", "Hyperactivity", "Impulsivity"}
```

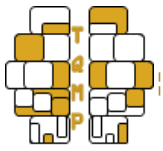


Figure 9 ■ List example.

```
In [12]: IQ_scores = [103, 87, 124, 95]
```

Figure 10 ■ Using indexing in a list.

```
In [14]: IQ_scores = [103, 87, 124, 95]
         IQ_scores[0]
Out[14]: 103
```

- If the answer is greater than 40, display “Correct”,
- Else, if the answer is below or equal to 40, display “Incorrect”
- Else, for any other value (strings, boolean, etc.), display “Invalid Format”

**Note:** You can have an infinite number of *if* statements by adding more *elif* and you can nest blocks of these statements inside other *if*, *elif* and *else* statements. That being said, the convention for the *if*, *elif* and *else* statements always follows this structure:

```
if (first conditional statement is True):
    Perform first desired action
elif (second conditional statement is True):
    if (sub conditional statement is True):
        Perform desired action
    else:
        if (sub of sub conditional statement is True):
            ...
    ...
elif (n conditional statement is True):
    Perform n desired action
else:
    Perform desired action (if none of the initial
condition are met)
```

**Note:** this is a great example of indentation as mentioned above.

## Loops

### While Loops

**While** loops in Python will execute a statement or a block of code as long as a set condition is met. This form of looping is useful when you do not know how many times you will need to perform a certain action before coding it. It is also beneficial if the number of times you will need to perform the action varies based on your data. The body of the **while** loop is defined with indentation. In order to code a **while** loop, start with the word “**while**” and your testing condition. Then, indent all of the code you want to run in the **while** loop (see Fig. 18). The **while** loop will continuously execute the code in the body of the loop as long as the condition you set is still met.

**Note:** There are two options to end an iteration early, by either jumping back to the beginning of the loop (**continue**) or jumping to the next section of code (**break**). When using loops, Python requires some entry in the body of a loop or a specific statement to tell it what to do. The **break** statements can be used to stop a loop and break out of it even if the condition set is still **True**. Using a **break** statement will make the program exit the loop and jump to the next section of code that follows it (see Fig. 18). By using the **continue** statements, you are ensuring that the current iteration stops but not by breaking out of the loop. Using **continue** allows you to jump back to the start of a

Figure 11 ■ Tuple example.

```
In [1]: participant_47 = (4.2461, True)
```



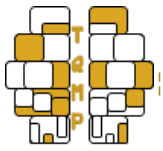


Figure 12 ■ Dictionary example

```
In [2]: ▶ Rodent_species = {}  
  
In [3]: ▶ Rodent_species["Mus musculus"] = True, "19 grams"  
         ▶ Rodent_species["Rattus norvegicus domestica"] = False, "450 grams"  
  
In [4]: ▶ Rodent_species  
Out[4]: {'Mus musculus': (True, '19 grams'),  
         'Rattus norvegicus domestica': (False, '450 grams')}
```

Figure 13 ■ Using keys to access values in a dictionary.

```
In [5]: ▶ Rodent_species["Mus musculus"]  
Out[5]: (True, '19 grams')
```

loop and continue iterating through your code. There are also *pass* statements. These act as placeholders that tell the program to do nothing. Essentially they are a null operation that fulfill the entry requirements of statements, loops or functions.

### For Loops

**For** loops in Python are used for iterating over a sequence, such as lists, dictionaries, strings, etc. They can be used to execute a series of statements a fixed number of times. As in *while* loops, the *break*, *continue*, and *pass* statements are all still very useful. Another useful function when using *for* loops is `range()`. The `range` function can be used in a *for* loop to execute the loop a specific amount of times. For instance, if a researcher wanted to iterate through a questionnaire, they could use a *for* loop and the `range` function to iterate a defined number of times over a list, as seen in Fig.19. The logic behind this example can be translated as follows: For each question ranging from 0 to 4 (remember, Python starts at 0 not 1) the *for* loop will print out the current iteration as the question number.

Loops can also be nested in order to perform more

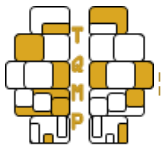
complex tasks. If we combine our two previous examples, we can nest a *for* loop within a *while* loop. In this example, a researcher has a survey that is incomplete. The *while* loop will continue until the survey is completed. Nested within this loop is a *for* loop that iterates through each question in the survey and prints the current question number. An *if* statement is then used to determine when the questionnaire has reached the fifth and final question (end of the *for* loop), thus completing the survey and ending the *while* loop.

### A Comprehensive Example

Now that you are familiar with the basics of Python, we can build upon our knowledge and create simple yet very powerful scripts. In this tutorial, we have explained many different concepts, but in order to summarize them and fully display the possibilities for your research in psychology, we will tackle a comprehensive example (see Fig. 21). The goal will be to create a general purpose corrector that can be used either for questionnaires, exams, or other similar tasks in psychology. In this particular example, we will use fictional results from the same-different task found in cog-

Figure 14 ■ Example of a block comment and an inline comment

```
In [1]: ▶ # IQ scores from group X  
         ▶ # Group X is comprised of students in the 11th grade  
         ▶ IQ_Scores = [103, 87, 124, 95]  
  
In [ ]: ▶ participant_47 = (4.2461, True) # reaction time in ms, control group
```

**Figure 15** ■ Example of conditional statements.

## Conditional Statements

'>' Is it greater than  
'<' Is it smaller than  
'>=' Is it greater or equal than  
'<=' Is it greater or smaller than  
'==' Is it equal/the same  
'!=' Is not equal/the same

```
print('is 5 > than 1: ', 5 > 1)
print('is 5 < than 1: ', 5 < 1)
print('is 2 >= than 2: ', 2 <= 2)
print('is 2 <= than 2: ', 2 >= 2)
print('is 10 equal to 5: ', 10 == 5)
print('is 10 not equal to 5: ', 10 != 5)
```

```
is 5 > than 1: True
is 5 < than 1: False
is 2 >= than 2: True
is 2 <= than 2: True
is 10 equal to 5: False
is 10 not equal to 5: True
```

nition. That being said, it is important to remember that this code could easily be adapted and expanded to accommodate any type of questionnaires, exams, or tasks.

To begin, we created a variable called *Answer\_key* and assigned a list containing the correct responses to the task to it. Then, we created a second variable called *Participant\_responses* and assigned it a list containing the participant's responses. Next, we created an empty list, called *Marking*, to be used to store the *Participant\_responses* as either a 1 for correct or 0 for incorrect. To ensure that we properly iterated/looped over all the answers, we created a variable called *Total\_trials* that was based on how many answers were in the *Answer\_key* (5) to use as one of the ending conditions for our *while* loop. Lastly we created a variable called *task\_complete* and gave it a *False* value so the *while* loop would run.

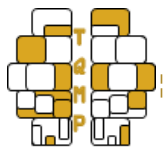
Once all variables were defined, we created our *while* loop. Directly inside of the *while* loop, we created a nested *for* loop that was set up to cycle through each trial and execute our desired operation. By using the *range()* function we defined that the *for* loop would execute the code up until the *Total\_trials* corresponding value (previously set to 5). To create our operation/conditional statement, we used the first *if* statement to check if the participant's response matched the answer in the answer key for that trial. If a match was found, the script would print the trial number followed by "Correct". To print the trial number, the *str()* function was used to convert the number into a string so it could be printed with the other strings. We then added the instruction to use the *.append()* function to add a "1"

to the *Marking* list. Our goal was to use the *Marking* list after the loops finished in order to calculate the average of correct responses. We also added an *else* for when the participant's answer was incorrect. If that was the case, the program would print the trial number followed by "Incorrect". We also added "0" to the *Marking* list by using *.append()*.

To verify if our code should end and display our results, we created an *if* statement that checked if the trial value was equal to the last index of our *Answer\_key*. If the loop had reached the last trial, it would change the *task\_complete* value from *False* to *True*, thus ending the *while* loop. To ensure that our *while* loop concluded its function, we added a print out a message saying that the same-different task was completed. As any good corrector, we wanted to add the possibility to see the calculated average of correct responses for a given participant. Thus, we created a variable called *Total\_correct* which contained the sums of all of the answers found in the *Marking* list. This was done by using the *sum()* function. The average was then calculated by dividing the *Total\_correct* variable by the *Total\_trials* variable and was stored in a new variable called *Avg\_Correct*. To ensure that the user would be able to see the participant's average as well as to thank them for using our program, we added two lines of code printing out the average and our message.

**Note:** As mentioned before, this code can be easily adapted to any similar tasks such as marking questionnaires or exams. It can also be scaled to iterate over a large number of participants or to handle other types of datasets.



**Figure 16** ■ Example of logical operators.

## Logical operators

- 'and' are both conditional statements True?
- 'or' are one of the statements True?
- 'not' are both statements False?

```
print('is 5 > 1 and 10 > 5: ', 5 > 1 and 10 > 5)
print('is 1 > 5 or 10 > 5: ', 1 > 5 or 10 > 5)
print('is 1 > 5 not true: ', not 1 > 5)
```

```
is 5 > 1 and 10 > 5: True
is 1 > 5 or 10 > 5: True
is 1 > 5 not true: True
```

**Figure 17** ■ Example of *if*, *elif* and *else* statements.

```
Participants_Answer = 42
```

```
if Participants_Answer > 40:
    print('Correct')
elif Participants_Answer < 40 or Participants_Answer == 40:
    print('Incorrect')
else:
    print('Invalid Format, please enter a number')
```

```
Correct
```

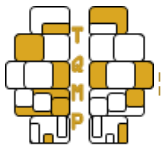
### So What's Next?

Now that you've learned the fundamentals of Python's programming language, it's time to let your imagination run wild and practice these concepts by creating your own scripts. The key to becoming fluent in any language is to speak and write as frequently as possible without any fear of failing. Thus, it is important to find interesting challenges that can motivate your brain and fingers to write code. A good place to start is with a Kaggle coding challenge, that can be found at [www.kaggle.com](http://www.kaggle.com). There, you can browse various challenges suited for any programmer, beginner or professional. Another possibility would be to look into your daily tasks and pinpoint anything that is being repeated exhaustively. In these situations, a simple *for* or *while* loop with certain conditional statements may save you time and allow you to automate these tasks. Regardless of your next step, the most important action to take is to practice with patience and to challenge yourself by div-

ing into more advanced Python functionality. For example, how to create functions, access libraries and incorporate sophisticated data analysis into your code. By doing so, you will start saving time in your research, impress your colleagues with innovative methodology and will have a new powerful tool to tackle any project.

### References

- American Psychiatric Association. (2000). Diagnostic and statistical manual of mental disorders-iv text revision, apa. *Washington, DC*.
- Harwani, B. (2012). *Introduction to Python programming and developing GUI applications with PyQt*. Nelson Education.
- Ngo, A. (2017). *Introduction to Python programming: Beginner to advanced, practical guide, tips and tricks, easy and comprehensive*.
- Pine, D. J. (2019). *Introduction to Python for science and engineering*. CRC Press.

**Figure 18** ■ Example of *while* loop and *break*.

```
In [3]: questionnaire = 0
while questionnaire < 5:
    print("One question complete")
    questionnaire = questionnaire + 1
print("Questionnaire completed")
```

One question complete  
One question complete  
One question complete  
One question complete  
One question complete  
One question complete  
Questionnaire completed

```
In [4]: questionnaire = 0
while questionnaire < 5:
    print("One question complete")
    questionnaire = questionnaire + 1
    break
print("Questionnaire completed")
```

One question complete  
Questionnaire completed

**Figure 19** ■ Example of *for* loop.

```
In [7]: for question in range(0,5):
        print('question number = ')
        print(question)
```

question number =  
0  
question number =  
1  
question number =  
2  
question number =  
3  
question number =  
4

Summerfield, M. (2010). *Programming in python 3: A complete introduction to the Python language*. Addison-Wesley Professional.

Van Rossum, G., & Drake, F. L. (2011). *The Python language reference manual*. Network Theory Ltd.

## Appendix

Table 2 shows the basic comparisons between the different variable types discussed in this tutorial. Ordered means that they have an order. In other words, they will always be displayed in the same order. Changeable (or mutable) means that the object can be changed after it is created. Indexing means the variable type has a built-in indexing system, such as an index key for each value. Duplicates means the variable type allows for duplicate members.

## Citation

Church, K., Rolon-Mérette, T., Ross, M., & Rolon-Mérette, D. (2021). Introduction to Python's syntax. *The Quantitative Methods for Psychology*, 17(1), S1–S12. doi:[10.20982/tqmp.17.1.S001](https://doi.org/10.20982/tqmp.17.1.S001)

Copyright © 2021, Church, Rolon-Mérette, Ross, and Rolon-Mérette. This is an open-access article distributed under the terms of the Creative Commons Attribution License (CC BY). The use, distribution or reproduction in other forums is permitted, provided the original author(s) or licensor are credited and that the original publication in this journal is cited, in accordance with accepted academic practice. No use, distribution or reproduction is permitted which does not comply with these terms.

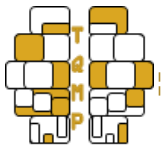


Figure 20 ■ Example of nested loops.

```

In [9]: survey_complete=False
while survey_complete==False:
    for question in range (0,5):
        print ('question number = ')
        print (question)
        if question==4:
            survey_complete=True
            print ('Survey Complete')

question number =
0
question number =
1
question number =
2
question number =
3
question number =
4
Survey Complete

```

Table 1 ■ Base colour coding for different IDEs.

	Python Shell	Jupyter	Spyder (Dark Theme)
<b>Variable</b>	Black	Black	White
<b>String</b>	Green	Dark Red	Salmon
<b>Function</b>	Purple	Dark Green	Light Orange
<b>Command</b>	Orange	Dark Green	Light Orange
<b>User Functions</b>	Blue	Black	Yellow
<b>Comment</b>	Dark Red	Light Green	Light Green
<b>Error Messages</b>	Light Red	Light Red	Light Red

Received: 11/08/2020 ~ Accepted: 23/09/2020

Table 2 ■ Overview of Variable Types and their properties.

Variable Types	Ordered	Changeable objects	Allows indexing	Allows duplicates
<b>List</b>	yes	yes	yes	yes
<b>Tuple</b>	yes	no	yes	yes
<b>Set</b>	no	yes	yes	no
<b>Dictionary</b>	no	yes	no	no

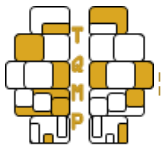


Figure 21 ■ Example incorporating many concepts from the tutorial.

```
In [25]: Answer_key = ['same', 'different', 'different', 'different', 'same']
Participant_responses = ['same', 'same', 'different', 'different', 'different']
Marking = []
Total_trials = 5
task_complete = False

while task_complete == False:
    for trial in range(0, Total_trials):
        if Participant_responses[trial] == Answer_key[trial]:
            print('trial ' + str(trial) + ' = Correct')
            Marking.append(1)
        else:
            print('trial ' + str(trial) + ' = Incorrect')
            Marking.append(0)
        if trial == 4:
            task_complete = True
            print('Same Different Task Complete')
            Total_correct = sum(Marking)
            Avg_correct = Total_correct / Total_trials
            print('Average of Correct responses = ' + str(Avg_correct))
            print('Thank you for participating')
```

```
trial 0 = Correct
trial 1 = Incorrect
trial 2 = Correct
trial 3 = Correct
trial 4 = Incorrect
Same Different Task Complete
Average of Correct responses = 0.6
Thank you for participating
```