Chapter 4 FAQ

PsychoPy Coder - Control

# You’ve got this

## You feel like you’re biking up a steep hill with a strong headwind?

You might experience that you never quite get the feeling that you are mastering the course. After comprehending what the solution is for an exercise, the next one turns out to be even more difficult than the previous one.

We realize that this might feel frustrating, but it is actually a sign that you are learning. Please don’t feel demotivated, embrace it! You’re going from relatively easy exercises in the text exercise to quite difficult exercises in the last class exercises. That’s actually a rather steep hill to climb! So every exercise is inevitably harder than the previous one. This is challenging for everyone who follows the course, so you are not alone. Keep on pushing through; we’re confident you all have what it takes to get to the end if you give it your time and effort. You’ve got this, keep on going!

## Are you just googling around while your script remains empty?

You might still be googling around aimlessly, not knowing how to start an exercise. After an hour of trying out a number of things you’re still stuck. Is this a recipe for certain disaster?

Absolutely not! You are simply spending time exploring the vocabulary and possibilities of Python, gradually acquiring an overview of the terrain. This is very valuable!

As an analogy, I often think about what my professor Italian once told me (after all, learning to program is like learning a new language). She went to Italy for a year when she was 18, without speaking a word of Italian. The first months, she just listened to her classmates and the members of the family where she was staying. Although she gradually started to understand more and more, she wasn’t able to formulate any sentences yet, which was very frustrating. Around Christmas she was about to give up, this wasn’t going anywhere. However, she stayed and in the month of January, all of a sudden things started to make sense. It was as if she broke through a barrier and was able to start speaking Italian. From there on, her learning curve really started to accelerate. Today, she teaches Italian at our university.

It may seem as if you are at a stand-still, but keep on practicing and exposing yourself to the vocabulary and possibilities of Python. Sooner or later, you’ll break through that barrier too!

# Debugging tips and tricks

## Print() it!

Again, printing is a very handy way to gain some insight into what a loop is doing. Especially in a complex combination of nested loops with if-statements.

Just insert a print statement in the loop that prints out the crucial variable on each pass through the loop. Doing this will help you understand where it goes wrong.

## Test new steps in a separate script

When you are working on a more complex script and you are adding new functionalities, it’s a good idea to first work out the new code that you want to insert in a separate file rather than inside the main script.

For instance, if you want to add the feedback to the Stroop task in CE 4.4, you can work out the if-statement in a separate file first before adding it to the script you already built. This will allow you to specifically and quickly test out only the code you are currently working on instead of having to go through the whole experiment.

A demo for CE 4.4 can be found [here](https://drive.google.com/file/d/1GMAdnnCzLdVaw4RskigUk7MpmrVowTjH/view?usp=sharing) (included in the folder with the solved Class Exercises for Chapter 4).

## Unpack complex lines of code

Let’s say you encounter a complex line of code in a script of one of your colleagues or in a demo script. How do you go about unpacking that code to understand it? Let’s demonstrate this with this demo code from CE 4.7:

| adjusted\_color = -((round(value\_array[i,f])/50)-1) |
| --- |

A good approach is to start at the most nested level of the code and start from there.

As you can see in the color-coded line below:

* we start with value\_array[i,f] in the most nested part of the code.
* Next, this is rounded with round().
* Then, the result is divided by 50 with ( /50)
* We subtract 1 from the result with ( -1)
* And finally, we reverse the sign with -

| adjusted\_color = -((round(value\_array[i,f])/50)-1) |
| --- |

This already helps, but it’s even better if we print out the values and rebuild the code from the inside out to see how it evolves.

A demo script on that build-up can be found [here](https://drive.google.com/file/d/15m1tA3kNfxwp10un8tb5svzsTuMIOACc/view?usp=sharing) (included in the folder with the solved Class Exercises for Chapter 4).

## Explain loops in slow motion

Say you have a (nested) loop structure that doesn’t do what you want it to. There is no error message, the loop just doesn’t do what you anticipated and you don’t get why. How can you gain insight into what is going on?

Let’s demonstrate this in this piece of code that was written for CE 4.3 (you can find the full script [here](https://drive.google.com/file/d/16MQHheP82ARch-GwRitf0QfqIkzTHJNK/view?usp=sharing), included in the folder with the solved Class Exercises for Chapter 4):

| # Loop over the ColorWords and FontColors  for i in range(len(ColorWord)):  Stroop\_stim = visual.TextStim(win, text = ColorWord[i])  Stroop\_stim.draw()  for i in range(len(ColorWord)):  Stroop\_stim = visual.TextStim(win, color = FontColor[i])  Stroop\_stim.draw()  win.flip()  time.sleep(0.5) |
| --- |

The question we got was something like: “I don’t understand what is going on, I’m drawing all my stimuli on the screen, but I only get the colors to change”. If you’re having difficulty understanding what goes wrong inside a loop, try to explain what is going on in the loop in slow motion. With slow motion we don’t mean something like this: “well, I loop over all the color words and over all the font colors, see?”. You have to go real slow and think through the individual iterations of the loops.

Let’s do this for the loops above:

* The first time that we pass through the loop, the index has the value 0.
* In the first line of code inside the loop, we access the first item in the list ColorWord, which is the word blue. The word blue is used as the text of the Stroop stimulus.
* In the second line of code inside the loop, the Stroop stimulus is drawn on the screen, but it isn’t made visible for the participant yet.
* Next, we start the second iteration of the for-loop.
* … Aha! In the first for-loop I just continue to draw the stimuli, but I’m not flipping and waiting! So my stimuli only get presented in the second for-loop.
* Also, I’m just changing the word of the Stroop stimulus over and over again, but I’m not changing the color before the stimulus is drawn!
* In the second for-loop I’m just changing the font colors, not the text of the stimulus.

## Learn some keyboard shortcuts

When learning to use control structures and especially nested control structures, using the right indentation is very important. Changing the indentation of a block of code by hitting the tab key on your keyboard for each line can be a tedious process. Luckily, there are keyboard shortcuts to increase or decrease the indentation level of an entire block of code in one movement.

Simply select the block of code that you want to move one indentation to the left or the right, go to Edit and you’ll see the *Indent selection* and *Dedent selection* options. Next to these options you can see the keyboard shortcuts that you can use.

# Additional explanation

## Looping over a string

In the text of Chapter 4 we already saw that we can loop over all the letters in a string like this:

| string = "hello world!"  for i in range(len(string)):  print(string[i]) |
| --- |

A string is simply a list of letters, so we can index it. To print all the letters, the for-loop needs to be executed for each letter and hence be len(string) long, taking on the indexes defined by range(len(string)).

This code is written in a way that helps you to gain insight into how for-loops work in general. However, we can loop over all the values in a string in a more direct way:

| for i in string:  print(i) |
| --- |

In this formulation of the for-loop, the index i takes on every value in the string. So on the first pass through the loop, the index i has the value ‘h’. In the second iteration, it has value ‘e’.

## CE 4.1: applying control as efficient as possible

Loops are used to avoid having to duplicate code in a script. However, loops also come with a cost, in the sense that they can start to slow down your code when you write inefficient loops. To illustrate this point of efficient loops, we go over a couple of implementations of CE 4.1. For the sake of illustration, we not only assume that you want to let the text “Happy halloween!” flash on and off on the screen, but that you also want to alternate the color of the eyes between black to red.

In a [first - most inefficient - implementation](https://drive.google.com/file/d/1lLw5owC-lBjazs5ioOfl5Tcsmm4vUziD/view?usp=sharing) (included in the folder with the solved Class Exercises for Chapter 4), we have placed a lot of code in the for-loop. This code does the job just fine: the color of the eyes alternates and the text flashes on the screen. However, we are doing a lot of unnecessary actions on the loop, such as setting the vertex coordinates and making the stimulus objects over and over again. When an object is reused multiple times, it suffices to make it once at the start of the script and just reuse it inside the loop. For instance by using .draw(). The object will still be there for you to use, no need to make it from scratch at each pass through the loop.

In the [second - already much more efficient - implementation](https://drive.google.com/file/d/1B45ed7gNdU7bCahjLDiTYW4OTVULMeCT/view?usp=sharing), we have placed most of the stimulus objects outside of the loop. However, we have kept the eyes inside of the loop because they need to alternate between black and red. Still, it is not necessary to remake the entire object from scratch, we only need to change the color property. This can be more easily done inside the loop via the .lineColor and .fillColor properties.

In the [third - even more efficient - implementation](https://drive.google.com/file/d/19ii8jERk_CCBVYwlW8uiXMkOUoX-KS4h/view?usp=sharing), we have placed the eye objects outside of the loop and update the .lineColor and .fillColor properties inside of the loop. Having moved all the visual objects outside of the loop will speed up your script a lot as making visual stimuli usually takes up some time. Still, the if-statement seems to contain some duplicated code. We can avoid this by first determining the color that needs to be filled in inside the if-statement, and then update the .lineColor and .fillColor properties outside of the if-statement.

The [fourth - most efficient - implementation](https://drive.google.com/file/d/1W3L-je9IO9bI54-CbV1oBnBrfGUw9S3H/view?usp=sharing) illustrates how also the if-statement can be scripted more efficiently.

## CE 4.6: Verifying trial characteristics

Some of you might wonder why we determine the correct response in advance or why we add the accuracy to the trial matrix. Why not just compare the color word to “red” inside the trial loop to determine whether the person is accurate and then display either *Correct!* or *Wrong answer!* as the immediate feedback message? The reason is mainly to avoid programming errors.

We determine the correct response in advance, because this allows us to easily verify whether our foreseen correct response is actually correct. The rules to determine what the correct responses are can become quite complicated and hence error prone. If you have to rerun your experiment every time to see whether the feedback is correct, this can take a lot of time (depending on the length of your trials or block structures in the experiment). Inspecting the first ten trials is no guarantee for a successful implementation of the response criteria in the rest of the trials. Determining the correct response for all the trials before the trial loop allows you to validate your approach for all trials in just a few seconds. You can find some additional inspiration on cross tables to verify the correct response implementation in [this scripts](https://drive.google.com/file/d/1hpG1Fp85YRitZcLPgCJaPJqbLUW_VQ1a/view?usp=sharing) (included in the folder with the solved Class Exercises for Chapter 4).

Likewise, you’ll want to analyse after the experiment whether the participant had a good or bad performance on the task, so why not store the accuracy of the participant while you are determining it for the feedback message? As a bonus, if you store it in an array you can inspect the performance of your trial loop after it is done by printing the trial matrix in the output window (we’ll learn later how to export it to an external file that you can open Excel or a statistical program such as R).

## CE 4.7: Explaining the code

Disclaimer: our solution for the bitcoin exercise is only one possible solution. You may have come up with a different and entirely valid version. The most important thing is that you practiced, made errors and googled around; all highly important programming skills! Coming up with a solution yourself is more important than trying to understand how our code came about. That being said, we of course made some implementation choices to demonstrate some programming tips and tricks. We’ll explain the reason behind our choices below.

### How to implement the color changing from white to red?

First of all, start with writing down what the color code for pure white and pure red looks like:

* White = (1,1,1)
* Red = (1,-1,-1)

So you see that the rgb code will have to evolve from white to red by lowering the values for green (2nd position) and blue (3rd position) from 1 to -1. It’s key to realize here that you won’t be changing the value for red (1st position), though this might have been your first hunch.

As both the green and the blue value vary in the same way, we only need one variable (*blue\_green* in my code).

The value of your bitcoins (denoted as *value* in my code) increases from 1 to 100. So we’ll have to translate that range from 1 to 100 to a range from 1 to -1 for *blue\_green*. To do so, I first brought the *value* range (with a span of 100) back to the same scale as the *blue\_green* range (spanning 2, the distance between -1 and 1). I also rounded *value* for the division to work smoothly.

numpy.ndarray.round(value)/50

The result will evolve from 0 to 2. Next, I subtracted 1 to make sure all results lie between -1 and 1 instead of between 0 and 2.

(numpy.ndarray.round(value)/50)-1

Finally, I reverse the scale, because now the *blue\_green* would be going from -1 to 1 as *value* increases from 1 to 100, while I want the *blue\_green* to go from 1 to -1.

-((numpy.ndarray.round(value)/50)-1)

That’s all, now we just have to assign this calculation to *blue\_green* and we are done.

blue\_green = -((numpy.ndarray.round(value)/50)-1)

Last but not least, I want to protect myself against the edge case that *blue\_green* evolves outside the range of -1 to 1, because the rgb code won’t accept such numbers. The main danger is that *blue\_green* will become smaller than -1 when *value* becomes larger than 100, so when *blue\_green* becomes smaller than -1 I bring it back to -1.

blue\_green = -((numpy.ndarray.round(value)/50)-1)

if blue\_green < -1:

blue\_green = -1

### Why is value\_array an array and why is values being appended to value\_array?

So we want to keep track of the entire history of the increasing value of your bitcoins. We need this history to plot out the increasing curve (see figure for CE 4.7).

To store all the temporary valuations that *value* takes on, I have created an array called *value\_array*. During each iteration of the while-loop, I add the current *value* to the *value\_array* by using the numpy.append() function (note that for the solution with the four bitcoins I used numpy.vstack()).

I could also have used a list, but lists are less suited for the last step of CE 4.7 (when you want to store the values across time for all four coins in a matrix). Of course I couldn’t use a tuple to store the history, because a tuple can’t be altered once it is made. Hence, an array is the most appropriate data type to use for *value\_array*.

Note that *value* is an integer/float and is constantly increasing. If I would have defined *value* as a tuple, I couldn’t have changed it in the while-loop, so a tuple is not the data type to go for.

### How to calibrate where the curve is being drawn on the screen?

Someone asked how I determined some of the numbers below to make sure the curve is drawn at the bottom of the screen:

## determine the start and stop value of all the lines that make up the curve

start\_y = (value\_array[i-1]/100) - 0.8

end\_y = (value\_array[i]/100) - 0.8

## draw this piece of the curve

curve.start = (-0.52 + 0.02\*i, start\_y)

curve.end = (-0.50 + 0.02\*i, end\_y)

curve.draw()

The division by 100 (e.g., value\_array[i-1]/100) is just to bring the values between 1 and 100 back to a scale from 0 to 1. Why? Because that matches with the part of the screen (with units set to "norm") that I want to use for the graph. To be more specific, I will use half of the screen: my plotting values will range between 0 and 1 while the normalized screen ranges from -1 to 1.

Next, I push the graph down to the lower part of the screen by subtracting 0.8, leaving 10% empty space at the bottom of the screen (0.2/2 = 10%). This is just personal taste.

Then, for the values on the horizontal axis. Out of experience with the algorithm I got a feeling for how many time steps were needed before *value* reaches 100. So I let the curve start at 1/4th of the width of the screen (measuring from left to right), hence the -0.50. Based on the time it takes for the algorithm to reach 100, I guessed that a step size of 0.02 would be just fine. Again, this is just a bit of tinkering to make it look pretty.

Ideally, of course, you write code that will automatically change the scale on the horizontal axis when you detect that it takes longer than estimated for *value* to reach 100.

Note that my solution is just one of multiple possible solutions and maybe not the most elegant or fail-proof one.

### Why we used a two-dimensional array

There are a number of reasons why you might like to use arrays instead of lists to solve the bitcoin exercise. This holds especially for the advanced challenge with the four bitcoins, but some reasons also apply to the version with only one bitcoin.

First, we will be using the bitcoin values as the basis to determine the vertical coordinates of the plot and the color gradient of the lines in the plotted curve. This will require applying some **mathematics** on the bitcoin values. As you know, the numpy arrays are written specifically with mathematics in mind. This allows us to perform all of the calculations necessary for the vertical coordinates, the horizontal coordinates and the color gradient in these few lines of code (at least when you have first determined all the values for the bitcoins, as demonstrated in [the step-by-step solution of the bitcoin exercise](https://drive.google.com/file/d/1bFyAUnAskqBU7qh0LqNHra2SJeQyfbhQ/view?usp=sharing), included in the folder with the solved Class Exercises for Chapter 4):

| # determine horizontal and vertical coordinates  nsteps = value\_array.shape[0]  y\_array = (value\_array/100) - 0.8  x\_array = -0.50 + (1/nsteps)\*numpy.array(range(nsteps))  # determining the color gradient  adjusted\_color\_array = -((value\_array/50)-1) |
| --- |

When you use lists, you don’t have the possibility of coding the solution in such an elegant line of code. You can only opt for a loop structure or use list comprehension (which is perfectly possible, but typically a bit harder to read). Hence, arrays usually result in shorter and more readable code.

Second, by using arrays you can store all the bitcoin values in one data structure, even when there are multiple bitcoins (one column per bitcoin, the rows are the time points). Those who used lists, were inclined to use four separate lists for the bitcoin values. This means that each line of code involving the list is quadrupled to accommodate all four lists. Think about what this would mean if you let hundred bitcoins increase simultaneously. This will result in a lot of **duplicated code**!

Of course you can make a list-of-lists. This would be a list in which each element is a list of for instance the current value of the four/hundred bitcoins. This is demonstrated below:

| bitcoin1 = 1  bitcoin2 = 2  bitcoin3 = 3  bitcoin4 = 4  # starting values of the bitcoins  Bitcoins =[[bitcoin1],[bitcoin2],[bitcoin3],[bitcoin4]]  # let one bitcoin update  Bitcoins[1].append(7)  print(Bitcoins)  # output: [[1], [2, 7], [3], [4]] |
| --- |

However, **printing out the values of the bitcoins in the output window** to see how your algorithm is performing, will result in one long list of values (as you can see in the copied output above). By comparison, when you print out the array, you get a structured output of rows and columns, giving you a much better overview of the various bitcoins (columns) and their evolution in time (rows). For instance, it will be trivial to scroll down in the output of the array to see the values at time point 47 and check which coin is ahead of the bunch at that moment. This is not trivial to look up in the output of the lists.

Third, you can use the dimensions of the array (e.g., number of rows and columns) to iterate over in a for-loop. You may have noticed that we referred to the number of column of the array to loop over all four bitcoins, and we referred to the number of rows in the array to loop over all the time points:

| ## for each of the four values  for f in range(value\_array.shape[1]):    ## draw evolving value  for i in range(1,value\_array.shape[0]): |
| --- |

Be careful however with the code you use to **access the number of dimensions** in the array!

* Although we could use the command .size to determine the number of elements in the one-dimensional values array in [step 3 of our solution](https://drive.google.com/file/d/1tdEzY_Dujuz61ykEm7GbRKwSsl434KMT/view?usp=sharing), .size is not a good option when working with more than one bitcoin. Why? Because .size gives you the total number of elements in the array (number of rows \* number of columns)! This was fine for the one-dimensional array because you only multiply with one column, but in the case of the four bitcoins you will quadruple the number of time points instead of just asking for the number of time points.
* Likewise, using len() to get the number of values of a one-dimensional array of bitcoin values is fine, but be careful when applying this code to a two-dimensional array: you’ll get the number of rows, but won’t be able to access the number of columns.
* Instead, use .shape[0] to get the number of rows (time points) and .shape[1] to get the number of columns (bitcoins).