Python for social and experimental psychology

Alexander Pastukhov

2020-10-27

Contents

Introduction

About the seminar

This is a material for *Python for social and experimental psychology* seminar. Each chapter covers a single seminar, introducing necessary ideas and is accompanied by a notebook with exercises, which you need to complete and submit. The material assumes no foreknowledge of Python or programming from the reader. Its purpose is to gradually build up your knowledge and allow you to create more and more complex games. Yes, games! Of course, the real research is about performing experiments but there is little difference between the two. The basic ingredients are the same and, arguably, experiments are just boring games. And, be assured, if you can program a game, you certainly can program an experiment.

We will start with simple *Guess a Number* text-only game with first you and then the computer doing the guessing. Next, we will implement a classic *Hunt the Wumpus* text adventure game that will require use of more complex structures. Once we master the basics, we will up the ante by making a video game with graphics and sounds using PsychoPy library to code a classic *Memory Game*. Finally, we will create a more dynamic game by making a clone of a *Flappy Bird*.

Remember that throughout the seminar you can and should(!) always ask me whenever something is unclear, you do not understand a concept or logic behind certain code. Do not hesitate to write me in the team or (better) directly to me in the chat (in the latter case, the notifications are harder miss and we don't spam others with our conversation).

You will need to submit your assignment one day before the next seminar (Tuesday before noon at the latest), so I would have time evaluate it and provide feedback.

As a final assignment, you will need to program a (currently mysterious) video game, which will only require the material covered by the seminar. Please inform me, If you require a grade, as then I will create a more specific description for you to have a clear understanding of how the program will be graded.

Note on exercises

In many exercises your will be not writing the code but understanding it. Your job in this case is "to think like a computer". Your advantage is that computers are very dumb, so instructions for them must be written in very simple, clear, and unambiguous way. This means that, with practice, reading code is easy for a human (well, reading a well-written code is easy, you will eventually encounter "spagetti-code" which is easier to rewrite from scratch than to understand). In each case, you simply go line-by-line, doing all computations by hand and writing down values stored in the variables (if there are too many to keep track of). Once you go through code in this manner, it will be completely transparent for you. No mysteries should remain, you should have no doubts or uncertainty about any(!) line. Moreover, then you can run the code and check that the values you are getting from computer match yours. Any difference means you made a mistake and code is working differently from how you think it does. In any case, **if you not 100% sure about any line of code, ask me, so we can go through it together!**

In a sense, this is the most important programming skill. It is impossible to learn how to write, if you cannot read first! Moreover, when programming you will probably spend more time reading the code and making sure that it works correctly than writing the new code. Thus, use this opportunity to practice and never use the code that you do not understand completely. So do use stackoverflow but do make sure you understand the code you copied!

Why Python?

The ultimate goal of this seminar is to teach you how to create an experiment for psychology research. There are many ways to achieve this end. You can use drag-and-drop systems either commercial like Presentation, Experiment Builder or free like PsychoPy Bulder interface. They have a much shallower learning curve, so you can start creating and running your experiments faster. However, the simplicity of their use has the price: They are fairly limited in which stimuli you can use and how you can control the presentation schedule, feedback, etc. Typically, they allow you to extend them by programming the desired behavior but you do need to know how to program to do this. Thus, I think that while these systems, in particular PsychoPy, are great tools to quickly bang a simple experiment together, they are most useful if you understand how they create the code and how you would program it yourself. Then, you will not fill being limited by the software, as you know you can program something the default drag-and-drop won't allow, but you can always opt in, if drag-and-drop is sufficient but faster. At the end, it is about having options and creative freedom to program an experiment that will answer your research question, not an experiment that your software allows you to program.

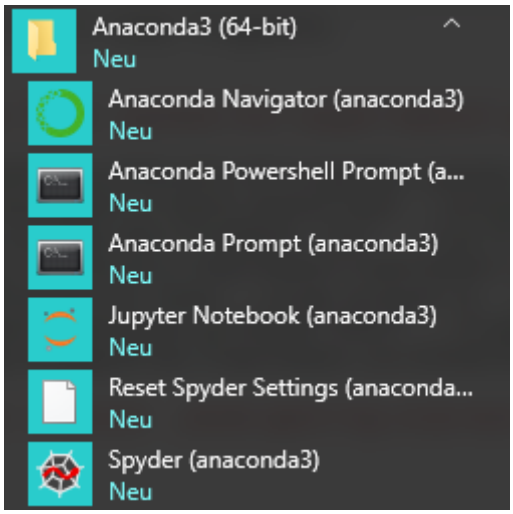
We will learn programming in Python, which is a great language that combines simple and clear syntax with power and ability to tackle almost any problem. The advantage of learning Python, as compared to say Matlab, which is commonly used in neuroscience, is that it allows you do almost anything. In this seminar, we will concentrate on desktop experiments but you can use it for online experiments (oTree), scientific programming (NumPy and SciPy), data analysis (pandas), machine learning (keras), website programming (django), computer vision (OpenCV), etc. Thus, learning Python will give you one of the most versitile programming tools that you can use for all stages of your research or work. And, Python is free, so you do not need to worry whether you or your future employer will be able to afford the license fees (a very real problem, if you use Matlab).

Getting Started

Installing Anaconda environment

First, install Anaconda, a Python distribution that includes many packages and tools out-of-the-box, makes it easy to install new packages and keep them updated. Follow this link and download the installer suitable for your platform. You can pick either 32- or 64-bit version but I would recommend the latter, so that we all have maximally similar setup (it won't really make a difference in practice, though). Follow the installer instructions and use defaults, unless you have reasons to modify them (e.g. folder location, as the drive for the default choice may have limited available space, as in my case).

After installation you will have a new *Anaconda3 (64-bit)* folder that contains links to programs.



You can use *Anaconda Navigator* that allows you to choose a specific programming environment, including Jupyter Notebook that we will use (not JupyterLab, it more versatile but we want to keep things simple at the beginning!).

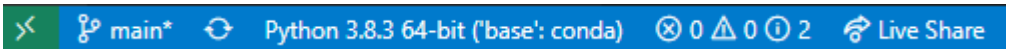
Alternatively, you can start *Jupyter Notebook* directly from the start menu. Please read the online documentation to familiarize yourself with Jupyter Notebook basic interface, e.g. how to create a new cell, run it, etc.

Installing Visual Studio Code

Visual Studio Code is a lightweight free open-source editor with strong support for Python. We will start use it in earnest, once our programs grow to be sufficiently long and complex. At the early stages, we will mostly use Jupyter notebooks and I would recommend using Jupyter notebooks using the default browser-based editor you installed as part of Anaconda. However, you can also work with Jupyter notebooks in VS Code directly.

As in case of Anaconda, download the installer for your platform and follow the instructions. Start VS Code and open any Python file, for example this one (use `Alt+click` to download it, ignore warnings, it is has only comments, so cannot harm you). When you open Python file for the first time, VS Code will suggest to install a Python extension. Do just that andinstall a linter as well when VS Code suggests that (linting highlights syntactical and stylistic problems in your code, making it easier to write consistent clear code).

Once the Python extension is activated, you will see which Python interpreter is used (you can have more than one or you may use have multiple virtual environments).



If the selected environment is the wrong one or you are simply not sure, click on it and it will open a drop-down list with all interpreters and environments you have. Consult VS Code online documentation on environments, if you need to change/add/delete environment (the exact settings may change, so looking at constantly updated online documentation is wiser than copying it here, so it would be outdated quite soon).

Installing PsychoPy

This step can wait until the first Memory Game seminar.

Download and install Standalone PsychoPy version. You can install PsychoPy as a conda package or via pip but using it as a standalone would ensure that you have all necessary additional libraries and a builder interface for the future use. We will use PsychoPy's python environment in VS Code.

Chapter 1

Python basics

Before we start, create a folder called *python-for-experiments* (or with some other more suitable but meaningful name) in you user folder (this is where Anaconda's Jupyter Notebook expect to find them). Download the exercise notebook and put it in this folder. Open Jupyter Notebook (see Getting Started, if you forgot how you do that), navigate to the folder you created and open the downloaded notebook. You will need to switch between explanations here and the exercises in the notebook, so keep them both open. The seminar contains the material introducing the concepts and tools that you require to implement the game step-by-step.

1.1 Variables

The first fundamental concept that we need to be acquainted with is **variable**. Variables are used to store information and you can think of it as a box with a name tag in which you can put something. The name tag on that the box is the name of the variable and its value what you store in it. For example, we can create a variable that stores the number of legs a game character has and we begin with a number typical for a human being.

FIGURE!

In Python, you would write

```
number_of_legs = 2
```

The **assignment statement** above has very simple structure `<variable-name> = <value>`. Variable name (name tag on the box) should be meaningful, it can start with letters or _ and can contain letters, numbers, and _ symbols but

must have no spaces in them. Preferably, you should use **snake_case** (all lower-case, underscore for spaces) to format your variable names. The `<value>` on the right side is a more complex story, as it can be simply hard-coded (as in example above), computed using other variables or the same variable, returned by a function, etc.

Using variables means that you can concentrate what corresponding values **mean** rather than worrying about what these values are. For example, the next time you need to compute something based on number of character's legs (e.g. how many pairs of shoes does a character need), you can compute it based on current value of `number_of_legs` variable rather than assume that it is `1`.

```
# BAD: why 1? Is it because the character has two legs or
# because we issue one pair of shoes per character irrespective of
# their actual number of legs?
pairs_of_shoes = 1

# BETTER!
pairs_of_shoes = number_of_legs / 2
```

Variables also gives you flexibility. Their values can change during the program run: player's score is increasing, number of lives decreasing, number of spells it can cast grows or falls depending on their use, etc. Yet, you can use the value in the variable to perform necessary computations. For example, here is a slightly extended `number_of_shoes` example.

```
number_of_legs = 2

# ...
# something happens and our character is turned into an octopus
number_of_legs = 8
# ...

# the same code still works and we know the correct number of pairs of shoes
pairs_of_shoes = number_of_legs / 2
```

As noted above, you can think about a variable as box labeled box you can store something in. That means that you can always "throw away" the old value and put something new. In case of variables, the "throwing away" part happens automatically, as the new value overwrites the old one. Check yourself, what will be final value of the variable in the code below? Go to exercise #1 in the notebook.

```
number_of_legs = 2
number_of_legs = 5
```

```
number_of_legs = 1
number_of_legs
```

As you have already seen, you can *compute* a value instead of specifying it. What would be the answer here? Do exercise #2 in the notebook.

```
number_of_legs = 2 * 2
number_of_legs = 7 - 2
number_of_legs
```

A **very important** rule that you must keep in mind when understanding assignments: the right side expression is evaluated first until the final value is computed, only when the computation finished the value is assigned to the variable (put in the box). What this means is that you can use the same variable on **both** sides! Let's take a look at this code:

```
x = 2
y = 5
x = x + y - 4
```

What happens when computer evaluates the following line?

```
x = x + y - 4
```

First, it takes *current* values of all variables (`x` and `y`) and substitutes them into the expression. After that internal step, the expression looks like

```
x = 2 + 5 - 4
```

Then, it computes the arithmetic expression for the right side and stores that new value in `x`

```
x = 3
```

Switch to the notebook and do exercise #3 to make sure you understand this.

1.2 Constants

Although the real power of variables is that you can change their value, you should use them even if the value remains constant. There are no constants in Python, rather the agreement is that their names should be all `UPPER_CASE`. So, if number of legs stays constant throughout the game, you should highlight that constancy and write

```
NUMBER_OF_LEGS = 2
```

I strongly recommend using constants and avoid hardcoding the values. First, if you have several identical values that mean different things (2 legs, 2 eyes, 2 ears, 2 vehicles per character, etc.), seeing a `2` in the code won't tell you what does this `2` mean (the legs? the ears? the score multiplier?). You can, of course, figure it out based on the code that uses this number but you could spare yourself that extra effort and use a constant instead. Then, you just read its name and the meaning of the value becomes apparent. Second, if you decide to *change* that value (say, our main character is now a tripod), when using a constant means you have only one place to worry about, the rest of the code stays as is. If you hard-coded that number, you are in for an exciting (not really) and long search-and-replace throughout the entire code.

1.3 Value types

So far, we only used integer numeric values (1, 2, 5, 1000…). Although, Python supports many different value types but we will concentrate on a small subset of them first:

- integer numbers, we already used, e.g. `-1`, `100000`, `42`.
- float numbers that can take any real value, e.g. `42.0`, `3.14159265359`, `2.71828`.
- strings that can store text. The text is enclosed between either paired quotes `"some text"` or apostrophes `'some text'`. This means that you can use quotes or apostrophes inside the string, as long as its is enclosed by the alternative. E.g., `"students' homework"` (enclosed in `"`, apostrophe `'` inside) or `'"All generalizations are false, including this one." Mark Twain'` (quotation enclosed by apostrophes).
- logical / Boolean values that are either `True` or `False`.

When using a variable it is important that you know what type of value it stores. In some cases, Python will automatically convert values between certain types, e.g. any integer value is also a real value, so conversion from `1` to `1.0` is mostly trivial and automatic. However, in other cases you may need to use explicit conversion. Go to exercise #4 and think about which code will run and which will throw an error due to incompatible types?

```
5 + 2.0
'5' + 2
'5' + '2'
'5' + True
5 + True
```

Surprised by the last one? This is because internally, `True` is also `1` and `False` is `0`!

You can explicitly convert from one type to another using special functions (we will explore these functions in greater detail below). For example, to turn a number or a logical value into a string, you simply write `str(<value>)`. In examples below, what would be the result (go to exercise #5 in the notebook)?

```
str(10 / 2)
str(2.5 + True)
str(True)
```

Similarly, you can convert to a logical/Boolean variable using `bool(<value>)` function. The rules are simple, for numeric values `0` is `False`, any other non-zero value is converted to `True`. For string, an empty string `''` is evaluated to `False` and non-empty string is converted to `True`. What would be the output in the examples below (go to exercise #6 in the notebook)?

```
bool(-10)
bool(0.0)

secret_message = ''
bool(secret_message)

bool('False')
```

Converting to integer or float numbers is trickier. The simplest case is from logical to integer/float, as `True` gives you `int(True)` is `1` and `float(True)` is `1.0` and `False` gives you `0`/`0.0`. When converting from float to integer, Python simply drops the fractional part (not rounding!). When converting a string, it must be a valid number of the corresponding type or the error is thrown. E.g., you can convert a string like `"123"` to and integer or a float but this won't work for `"a123"`. Moreover, you can convert `"123.4"` to floating-point number but not to an integer, as it has fractional part in it. Given all this, which cells would work and what output would they give (go to exercise #7 in the notebook)?

```
float(False)
int(-3.3)
float("67.8")
int("123+3")
```

1.4 Printing output

To print the value, you need you use `print()` function (we will talk about functions in greater detail later). In simplest case, you pass the value and it will

be printed out.

```
print(5)
```

```
## 5
```

or

```
print("five")
```

```
## five
```

Of course, you already know about the variables, so rather that putting the value directly, you can pass a variable instead and it is its value that will be printed out.

```
number_of_pancakes = 10
print(number_of_pancakes)
```

```
## 10
```

or

```
breakfast = "pancakes"
print(breakfast)
```

```
## pancakes
```

You can also pass more than one value/variable to the print function and all the values will be printed one after another. For example, if we want to tell the user what did I had for breakfast and just how many of those, we can do

```
breakfast = "pancakes"
number_of_items = 10
print(breakfast, number_of_items)
```

```
## pancakes 10
```

Go to exercise #8 and figure out what will be printed by the code below:

```
dinner = "stake"
count = 4
desert = "cupcakes"

print(count, dinner, count, desert)
```

However, you probably would want to be more explicit, when you print out the information. For example, imagine you have these three variables:

```
meal = "breakfast"
dish = "pancakes"
count = 10
```

You could, of course do `print(meal, dish, count)` but it would be nicer to print

"*I had* ***10 pancakes*** *for* ***breakfast***"

there items in bold would be the inserted variables' values. For this, we need to use string formatting. Please note that the string formatting is not specific to printing, you can create a new string value via formatting and store it in a variable (without printing it out) or print it out (without storing it).

1.5 String formatting

A great resource on string formatting in Python is pyformat.info. As Python constantly evolves, it now has more than one way to format strings. Below, I will introduce the "old" format that is based on classic string formatting used in `sprintf` function is C, Matlab, R, and many other programming languages. It is somewhat less flexible than a newer ones but for simple tasks the difference is negligible. Knowing the old format is useful because of its generality. If you want to learn alternatives, read at the link above.

The general call is `"a string with formatting"%(tuple of values to be used during formatting)`.

In `"a string with formatting"`, you specify where you want to put the value via `%` symbol that is followed by an *optional* formatting info and the *required* symbol that defines the **type** of the value. The type symbols are * `s` for string * `d` for an integer * `f` for a float value, so that it is fully printed * `g` for an optimally printed float value, so that scientific notation is used for large values (*e.g.*, `10e5` instead of `100000`).

Here is an example of formatting a string using an integer:

```
print("I had %d pancakes for breakfast"%(10))
```

```
## I had 10 pancakes for breakfast
```

You are not limited to a single value that you can put into a string. You can specify more locations via `%` but you must make sure that you pass the right number of values. Before running it, can you figure out which call will actually work (and what will be the output) and which will produce an error (go to exercise #9)?

```
print('I had %d pancakes and either %d  or %d stakes for dinner'%(2))
print('I had %d pancakes and %d stakes for dinner'%(7, 10))
print('I had %d pancakes and %d stakes for dinner'%(1, 7, 10))
```

In case of real values you have two options: `%f` and `%g`. The latter uses scientific notation (e.g. `1e10` for `10000000000`) to make the representation more compact. To get a better feeling for the difference do exercise #10.

These is much more to formatting and you can read about it at pyformat.info. However, these basics are sufficient for us to start programming our first game during the next seminar. Don't forget to submit your exercise notebook and see you next time!

Chapter 2

Guess the Number

2.1 Game description

We will program a game where one participant (computer) picks the number within a certain range, say, between 1 and 10 and the other participant (player) is trying to guess it. After every guess, the first participant (computer) responds whether the actual number is lower than a guess, higher than a guess, or matches it. The game is over when the player correctly guess the number or (in the later version of the game) runs out of attempts.