Python for social and experimental psychology

Alexander (Sasha) Pastukhov

2020-11-07

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 that 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 a simple *Guess a Number* text-only game in which first you and then the computer will be 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 *video* games with graphics and sounds using PsychoPy library. We will start with a classic *Memory Game* and, then, 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 or 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 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 reading and 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 "spaghetti-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, reading the code 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. This means that you certainly can 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 a price: They are fairly limited in which stimuli you can use and how you can control the presentation schedule, conditions, 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 underlying code and how you would program it yourself. Then, you will not be 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.

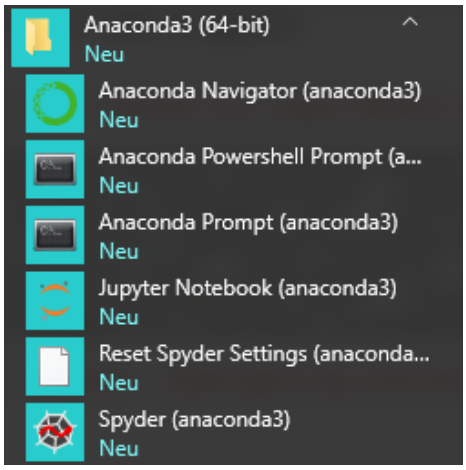
We will learn programming in Python, which is a great language that combines simple and clear syntax with power and ability to tackle almost any problem. The advantage of learning Python, as compared to say Matlab, which is commonly used in neuroscience, is that it allows you do almost anything. In this 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. 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 is more versatile but we want to keep things simple at the beginning!).

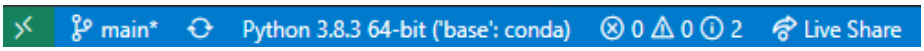
Alternatively, you can start *Jupyter Notebook* directly from the start menu. Please read the online documentation to familiarize yourself with Jupyter Notebook basic interface, e.g. how to create a new cell, run it, etc.

Installing Visual Studio Code

Visual Studio Code is a free lightweight 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 that and install a linter 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 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).

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. However, 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 prepackaged PsychoPy's python environment in VS Code.

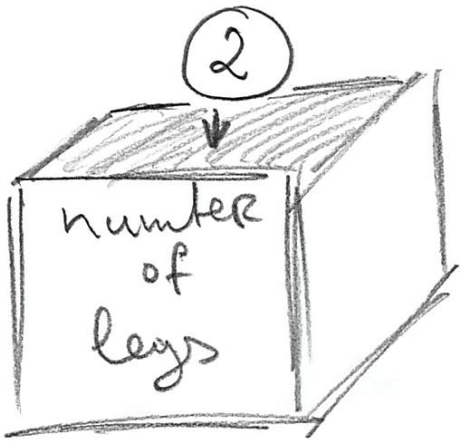
Chapter 1

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 expects 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.

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, so that you can put something into it. The name tag on that 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. We begin with a number typical for a human being.



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 _ symbol but not spaces, tabs, special characters, etc. Python recommends (well, actually, insists) that you 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 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 give 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 always 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 still can compute the correct number of pairs of shoes
pairs_of_shoes = number_of_legs / 2
```

As noted above, you can think about a variable as a 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?

```
number_of_legs = 2
number_of_legs = 5
number_of_legs = 1
number_of_legs
```

Do exercise #1.

As you have already seen, you can *compute* a value instead of specifying it. What would be the answer here?

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

Do exercise #2.

1.2 Assignments are not equations!

Very important: although assignments *look* like mathematical equations, they are **not equations!** Assignments follow 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, then and only then the final value is assigned to the variable specified on the left side (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 last line?

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

First, it takes *current* values of all variables (`2` for `x` and `5` for `y`) and substitutes them into the expression. After that internal step, the expression looks like

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

Then, it computes the expression on the right side and, **once the computation is completed**, stores that new value in `x`

```
x = 3
```

Do exercise #3 to make sure you understand this.

1.3 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 true constants in Python, rather an agreement that their names should be all `UPPER_CASE`. Accordingly, when you see `SUCH_A_VARIABLE` you know that you should not change its value. Technically, this is just a recommendation, as no one can stop you from modifying value of a `CONSTANT`. However, much of Python's ease-of-use comes from such "gentlemen's agreements" (such as `snake_case` convention above), which you should respect. We will encounter more of them when learning about objects.

Taking all this into account, 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 will not 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 (and it is the meaning not the actual value that you are mostly interested in). 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) but definitely long search-and-replace throughout the entire code.

Do exercise #4.

1.4 Value types

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

- 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). There is much much more to strings and we will cover that material throughout the course.
- 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 and this is mostly on you. Python will raise an error, if you try doing a computation using incompatible. 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 #5 and try guessing 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
```

Do exercise #5.

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. 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?

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

Do exercise #6.

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?

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

secret_message = ''
bool(secret_message)

bool('False')
```

Do exercise #7.

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 generated. 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 produce?

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

Do exercise #8.

1.5 Printing output

To print the value, you need you use `print()` function (we will talk about functions in general later). In the 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 than putting a value directly, you can pass a variable instead and its value 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
```

What will be printed by the code below?

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

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

Do exercise #9.

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***", where 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).

1.6 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
- `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?

```
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))
```

Do exercise #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 a representation more compact.

Do exercise #11 to get a better feeling for the difference.

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

Seminar #1 covered Python basics, so now you are ready to start developing you first game! We will build it step by step and there will be a lot to learn about input, libraries, conditional statements, and indentation. As before, download exercise notebook, copy it in your designated folder, and open it in Jupyter Notebook.

2.1 Game description

We will program a game in which 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.

Our first version will allow just one attempt (will make it more fun later on) and the overall game algorithm will look like this:

```
# 1. computer generates a random number
# 2. prints it out for debug purposes
# 3. prompts user to enter a guess
# 4. compares two numbers and print outs the outcome
#    "My number is lower", "My number is higher", or "Spot on!"
```

2.2 Let's pick the number (Exercise 1)

Let us start by creating a variable that will hold a number that computer "picked". Let us name it `number_picked` (you can some other meaningful name as well but it might be easier if we all stick to the same name). To make things a bit simpler at the beginning, let us assign some hard=coded arbitrary number between 1 and 10 to it (whatever you fill like). Then, let us print it out, so that we know the number ourselves (we know it now but that won't be the case when computer will generate it randomly). Use string formatting to make things user-friendly, e.g., print out something like "The number I've picked is ...". Your code should be a two-liner:

```
# 1. create variable and set it value
# 2. print out the value
```

Put your code into exercise #1 and make sure your code works!.

2.3 Asking user for a guess (Exercise 2)

Now we need to ask the player to enter their guess. For this, we will use input([prompt]) function (here and below the links lead to the official documentation). It prints out `prompt` (a string) if you supplied it, reads the input (key presses) until the user presses `Enter`, and returns it **as a string**. For a moment, let us assume that the input is always an valid integer number (so, type only valid integers!), so we can convert it to an integer without extra checks (will add them later) and assign this value to a new variable called `guess`. Thus, you need to write a single line assignment statement with `guess` variable on the left side, whereas on the right should be a call to the `input(...)` function (think of a nice prompt message) wrapped by the type-conversion to `int(...)`. Switch to exercise 2 and, for the moment, only enter valid integers when running the code, so that the conversion works without an error.

Put your code into exercise #2.

2.4 Conditional *if* statement

Now we have two numbers: One that computer picked and one that is player's guess. Now, we need to compare them to provide correct output message. For this, we will use conditional if statement:

```
if some_condition_is_true:
    # do something
```

```
elif some_other_condition_is_true:
    # do something else
elif yet_another_condition_is_true:
    # do yet something else
else:
    # do something only if all conditions above are false.
```

Only the `if` part is required, whereas `elif` (short for "else, if") and `else` are optional. Thus you can do something, only if a condition is true:

```
if some_condition_is_true:
    # do something, but OTHERWISE DO NOT DO ANYTHING
    # and continue with code execution

# some code that is executed after the if-statement,
# irrespective of whether the condition was true or not.
```

Before we can properly use conditional statements, you need to understand (1) the conditions themselves and (2) use of indentation as a mean of grouping statements together.

2.5 Conditions and comparisons (exercises 3-8)

Condition is any expression that can be evaluated to see whether it is `True` or `False`. A straightforward example of such expression are comparisons, in human language expressed as "is today Thursday?", "is the answer equal to 42", "is it raining and I have an umbrella?". We will concentrate on them here but later you will see that in Python **any** expression is either `True` or `False`, even when it does not look like a comparison.

For the comparison, you can use the following operators:

- *"A is equal B"* is written as `A == B`.
- *"A is not equal B"* is written as `A != B`.
- *"A is greater than B"* and *"A is smaller than B"* are, respectively, `A > B` and `A < B`.
- *"A is greater than or equal to B"* and *"A is smaller than or equal to B"* are, respectively, `A >= B` and `A <= B` (please note the order of symbols!).

Go to exercise #3 to solve some comparisons.

You can *invert* the logical value using `not` operator, as `not True` is `False` and `not False` is `True`. This means that `A != B` is the same as `not A == B` and, correspondingly, `A == B` is `not A != B`. To see how that works, consider both cases when `A` is indeed equal `B` and when it is not.

- If A is equal B then `A == B` evaluates to `True`. The `A != B` is then `False`, so `not A != B` → `not False` → `True`.
- If A is not equal B then `A == B` evaluates to `False`. The `A != B` is then `True`, so `not A != B` → `not True` → `False`.

Go to exercise #4 to explore this inversion yourself.

You can also combine several comparisons using `and` and/or `or` operators. As in human language, `and` means that both parts must be true: `True and True` → `True` but `True and False` → `False`, `False and True` → `False`, and `False and False` → `False`. Same holds if you have more have than two conditions/comparisons, **all** of them must be true. In case of `or` only one of the statements must be true, e.g. `True and True` → `True`, `True and False` → `True`, `False and True` → `True`, but `False and False` → `False`. Again, for more than two comparisons/conditions at least one of them should be true for the entire expression to be true.

Do exercises #5 and #6.

Subtle but important point: conditions are evaluated from left to right until the whole expression can be definitely resolved. This means that if the first expression in a `and` pair is `False`, the second one is **never evaluated**. I.e., if `first and second` expressions both need to be `True` and you know that already `first` expression is false, the whole expression will be `False` in any case. This means that in the code below there will be no error, even though evaluating `int("e123")` raises `ValueError`.

```
2 * 2 == 5 and int("e123") == 123
```

However, reverse the order, so that `int("e123") == 123` needs to be evaluated first and you get the error message

```
int("e123") == 123 and 2 * 2 == 4
# Generates ValueError: invalid literal for int() with base 10: 'e123'
```

Similarly, if *any* expression in `or` is `True`, you do not need to check the rest.

```
2 * 2 == 4 or int("e123") == 123
```

However, if the first condition is `False`, we do need to continue (and stumble into an error):

```
2 * 2 == 5 or int("e123") == 123
# Generates ValueError: invalid literal for int() with base 10: 'e123'
```

Do exercise #7.

Finally, like in simple arithmetic, you can use brackets `()` to group conditions together. Thus a statement "I always eat chocolate but I eat spinach only when I am hungry" can be written as `food == "chocolate" or (food == "spinach" and hungry)`. Here, `the food == "chocolate"` and `food == "spinach" and hungry` are evaluated independently, their values are substituted in their place and then the `and` condition is evaluated.

Do exercise #8.

2.6 Grouping statements via identation (exercise #9)

Let us go back to the conditional if-statement. Take a look at following code example, in which statement #1 is executed only if some condition is true, whereas statement #2 is executed after that irrespective of the condition.

```
if some_condition_is_true:
    statement #1
statement #2
```

Both statements #1 and #2 appear after the if-statement, so how does Python now that the first one is executed only if condition is true but the other one always runs? The answer is indentation (the **4 (four!)** spaces, they are automatically added whenever you press `Tab` and removed whenever you press `Shift + Tab`) that puts statement #1 *inside* the if-statement. Thus, indentation shows whether statements belong to the same group (same indentation as for `if` and `statement #2`) or are inside conditional statement, loop, or function (`statement #1`). For more complex code that will have, for example, if-statement inside an if-statement inside a loop, you will express this by adding more levels of indentation. E.g.

```
# some statements outside of the loop (0 indentation)
while game_is_not_over: # (0 indentation)
    # statements inside of the loop
    if key_pressed: # (indentation of 4)
        # inside loop and if-statement
        if key == "Space": # (indentation of 8)
            # inside the loop, and if-statement, and another if-statement
            jump() # (indentation of 12)
        else: # (indentation of 4)
            # inside the loop, and if-statement, and else part of another if-statement
            stand() # (indentation of 12)
```

```
    # statements inside of the loop but outside of the outermost if-statement
    print(key) # (indentation of 4)

# some statements outside of the loop (0 indentation)
```

Pay very close attention to the indentation as it determines which statements are executed together!

Do exercise #9.

The `if` and `ifelse` statements are evaluated until one of them turns out to be `True`. After that any following `ifelse` and `else` statements are simply ignored.

Do exercise #10.

2.7 Checking the answer (Exercise 11)

Now you have all necessary instruments to finish the first version of our game. Go to exercise #11 and, first, copy-paste your solutions to exercise #1 (settings computer pick and printing it out) and #2 (getting player input as an integer). Now, add conditional statements below, so that

- if the computer pick is smaller than player's guess, it will print `"My number is lower!"`
- if the computer pick is larger than player's guess, it will print `"My number is higher!"`
- if two numbers are identical, it will print `"Spot on!"`

Put your code into exercise #11.

2.8 Picking number randomly (Exercise 12)

Our game is "feature-complete": computer picks a number, player makes a guess, computer responds appropriately. However, currently we are playing for both sides, as we hand pick the number for computer. Now, we will let computer pick this number itself using randint(a, b) function. It is part of the random library, so you will need to *import* it first. We will talk about libraries and importing them in greater detail later. For now, it suffices that the top line of your code is

```
from random import randint
```

Function `randint(a, b)` generates a random integer on the interval `a..b`. In our case, this interval is `1..10`. Go to exercise #11. First copy-paste your solution for exercise #12. Add the `from random import randint` as the first line. Then, replace the hard-coded value you used for computer's pick with a call to `randint()` function. Run the code several times to check that computer does pick different random values.

Put your code into exercise #12.

Congratulations, you just programmed your first computer game! Yes, it is very simple but it has key ingredients: a random decision by computer, user input, and feedback. Next time, you will learn about loops to allow for multiple attempts and about functions to make your code modular and reliable. In the meantime, let us solidify your knowledge by programming yet another game!

2.9 One-armed bandit (Exercise 13)

You know everything you need to program a simple version of an "one-armed bandit" game (exercise #13). Here is the game logic:

1. `from random import randint`
2. Generate three random integers (say, between 1 and 5) and store them in three variables `slot1`, `slot2`, and `slot3`.
3. Print out the numbers, use string formatting to make it look nice.
4. In addition,
    - if all three values are the same, print `"Three of a kind!"`.
    - If only two numbers match, print `"Pair!"`.
    - Print nothing, if all numbers are different.

Put your code into exercise #13.

Chapter 3

Guess the Number, the Sequel

During our previous seminar, you programmed a single-attempt-only "Guess the Number" game. Now, you will expand to multiple attempts and will add other bells-and-whistles to make it more fun. Download the exercise notebook before we start!

3.1 While loop (Exercises 1-2)

If you want to repeat something, you need to use loops. There are two types of loops: while loop, which is repeated *while* a condition is true, and for loop that iterates over items (we will use it later).

The basic structure of a *while* loop is

```
# statements before the loop

while <condition>:
    # statements inside are executed
    # repeatedly for as long as
    # the condition is True

# statements after the loop
```

The `<condition>` here is any expression that is evaluated to be either `True` or `False`, just like in an *if-elif-else* conditional statement.

Do exercise #1.

Let us use *while* loop, so that the player keeps guessing until finally getting it right. You can copy-paste the code you programmed during the last seminar or could redo it from scratch (I would strongly recommend you doing the latter!). The overall program structure should be the following

```
from random import randint

# generated random number and store in computer_pick variable
# print it out for debugging purposes
# get player input, convert it to an integer, and store

# while <players guess is not equal to the value the computer picked>:
    # print out "my number is smaller" or "my number is larger" using if-else statemen

# print "Spot on!" (because if we got here that means guess is equal to the computer's
```

Put your code into exercise #2.

3.2 Counting attempts (Exercise #3)

Now let us add a variable that will count the total number of attempts the player required. For this, create a new variable (call it `attempts` or something similar) *before the loop* and initialize it `0`. Add `1` to it every time the player inputs the guess. After the loop, expand the `"Spot on!"` message you print out by adding information about the attempts count. Use string formatting to make things look nice, e.g. `"Spot on, you needed 5 attempts"`.

Put your code into exercise #3.

3.3 Breaking (and exiting, Exercise #4)

While loop is continuously executed while the condition is `True` and, importantly, all code inside is executed before the condition is evaluated again. However, sometimes you may need to abort sooner without executing the remaining code. For this, Python gives you a `break` statement that causes the program to exit the loop immediately and to continue with the code after the loop.

```
# this code runs before the loop

while <somecondition>:
  # this code runs on every iteration

    if <someothercondition>:
```

```
        break

  # this code runs on every iteration but not when you break out of the loop

# this code runs after the loop
```

Do exercise #4 to build the intuition.

3.4 Limiting number of attempts via break (Exercise 5)

Let's put the player under some pressure! Decide on maximal number of attempts allowed and stores in a constant. Pick an appropriate name (e.g. `MAX_ATTEMPTS`) and REMEMBER, ALL CAPITAL LETTERS for a constant name! Now, use `break` to quit the `while` loop, if current attempt number is greater than `MAX_ATTEMPTS`.

Put your code into exercise #5.

3.5 Correct end-of-game message (Exercise 6)

Think about the final message. Currently it says "Spot on…" because we assumed that you exited the loop because you gave the correct answer. With limited attempts that is not the case, as the player could out of the loop because

1. They answered correctly
2. They ran out of attempts.

Use `if-else` conditional statement to print out an appropriate message (e.g., `"Better luck next time!`, if the player lost).

Put your code into exercise #6.

3.6 Limiting number of attempts with a break (Exercise 7)

Modify your code to work without `break` statement. Modify your condition so that loop repeats while player's guess is incorrect and the number of attempts is still less than the maximally allowed.

Put your code into exercise #7.

3.7 Show remaining attempts (Exercise 8)

Modify the `input` prompt message to include number of *remaining* attempts. E.g. `"Please enter the guess, you have X attempts remaining"`.

Put your code into exercise #8.

3.8 Repeating the game (Exercise 9)

Let us an option for the player to play again. This means putting *all* the current code inside of another `while` loop that is repeated for as long as the player wants to play. The code should look following:

```
from random import randint

# define MAX_ATTEMPTS

# define a variable called "want_to_play" and set to True
while want_to_play:
  # your working game code goes here ()

  # ask user whether via input function. E.g. "Want to play again? Y/N"
  # want_to_play should be True if user input is equal to "Y"

# very final message, e.g. "Thank you for playing the game!"
```

Pay extra attention to indentation to group the code properly!

Put your code into exercise #9.

3.9 Best score (Exercise 10)

A "proper" game typically keeps the track of players' performance. Let us record what was the fewest number of attempts that the player needed to guess the number. For this, create a new variable `fewest_attempts` and set it to `MAX_ATTEMPTS` (this is as bad as the player can be). Think, where do you need to create it? Once a game round is over and you know how many attempts the player required, update it if the number of attempts that the player used was *less* than the current value. You can add the information about "Best so far" into the game-round-over message.

Put your code into exercise #10.

3.10 Counting game rounds (Exercise 11)

Let us count how many rounds the player played. The idea and implementation is the same as with counting the attempts. Create a new variable, initialize it to 0, increment by 1 whenever a new round starts. Include the total number of rounds into the very final message, e.g. "Thank you for playing the game X times!"

Put your code into exercise #11.

3.11 Wrap up

Most excellent, you now have a proper working computer game with game rounds, limited attempts, best score, and what not!

Chapter 4

Hunt the Wumpus, part 1

We will program text adventure computer game Hunt the Wumpus: "In the game, the player moves through a series of connected caves, arranged in a dodecahedron, as they hunt a monster named the Wumpus. The turn-based game has the player trying to avoid fatal bottomless pits and"super bats" that will move them around the cave system; the goal is to fire one of their "crooked arrows" through the caves to kill the Wumpus…"

As before, we will start with a very basic program and will build it step-by-step towards the final version. Don't forget to download the exercise notebook.

4.1 Lists

So far, we were using variables to store single values: computer's pick, player's guess, number of attempts, etc. However, you can store multiple values in a variable using lists. The idea is fairly straightforward, a variable is not a simple box but a box with slots for values numbered from `0` to `len(variable)-1`.

The list is defined via square brackets `<variable> = [<value1>, <value2>, ... <valueN>]` and an individual value can be accessed also via square brackets `<variable>[<index>]` where index goes from `0` to `len(<variable>)-1` (`len(<object>)` function returns number of items in an object, in our case, it would be a list). Thus, if you have five values in the list, the index of the first one is `0` (not `1`) and the index of the last one is `4` (not `5`)!

Do exercise #1 see how lists are defined and indexed.

You can also get many values from the list via so called *slicing* when you specify index of many elements via `<start>:<stop>`. There is a catch though and, as this is a recurrent theme in Python, pay close attention: The index slicing builds goes from `start` up to **but not including** `stop`, in mathematical notation

[*start*, *stop*). So, if you have a list `my_pretty_numbers` that holds five values and you want to get values from second (index `1`) till fourth (index `3`) you need to write the slice as `1:4` (not `1:3`!). This *including the start but excluding the stop* is both fairly counterintuitive (I still have to consciously remind myself about this) and widely used in Python.

Do exercise #2 to build the intuition.

You can also omit either `start` or `stop`. In this case, Python will assume that a missing `start` means `0` (the index of the first element) and missing `stop` means `len(<list>)`. If you omit *both*, e.g., `my_pretty_numbers[:]` it will return all values, as this is equivalent to `my_pretty_numbers[0:len(my_pretty_numbers)]`.[1]

Do exercise #3.

You can also use *negative* indexes that are relative to length of the list. Thus, if you want to get the *last* element of the list, you can say `my_pretty_numbers[len(my_pretty_numbers)-1]` or just `my_pretty_numbers[-1]`. The last-but-one element would be `my_pretty_numbers[-2]`, etc. You can use negative indexes for slicing but keep in mind *including the start but excluding the stop*: `my_pretty_numbers[:-1]` will return all but last element of the list not the entire list.

Do exercise #4.

The slicing can be extended by specifying a `step`, so that `stop:start:step`. This can be combined with omitted and negative indexes. To get every *odd* element of the list, you write `my_pretty_numbers[::2]`:

```
my_pretty_numbers = [1, 2, 3, 4, 5, 6, 7]
my_pretty_numbers[::2]
```

```
## [1, 3, 5, 7]
```

Do exercise #5.

Finally, for those who are familiar with R, the good news is that Python does not allow you to use indexes outside of the range, so trying to get 6^{th} element (index `5`) of a five-element-long list will generate a simple and straightforward error (a so-called fail-fast principle). The bad news is that if your *slice* is larger than the range, it will truncated to the range without an extra warning or an error. So, for a five-element list `my_pretty_numbers[:6]` will return all numbers of to the maximal possible index (thus, effectively, this is equivalent to `my_pretty_numbers[:]`). Moreover, if the slice is empty (`2:2`, cannot include `2`, even though it starts from it) or the entire slice is outside of the range, Python will return an empty list, again, neither warning or error is generated.

[1] Note, that this is almost but not quite the same thing as just writing `my_pretty_numbers`, the difference is subtle but important and we will look into it later when talking about mutable versus immutable types.

Do exercise #6.

4.2 Caves

In our game, the player will wander through a systems of caves with cave being connected to three other caves. The cave layout will be *CONSTANT*, so we will define at the beginning of the program as follows.

```
CAVES = [[1, 4, 5], [2, 0, 7], [3, 1, 9], [4, 2, 11],
         [0, 3, 13], [6, 14, 0], [7, 5, 15], [8, 6, 1],
         [9, 7, 16], [10, 8, 2], [11, 9, 17], [12, 10, 3],
         [13, 11, 18], [14, 12, 4], [5, 13, 19], [16, 19, 6],
         [17, 15, 8], [18, 16, 10], [19, 17, 12], [15, 18, 14]]
```

Let us decipher this. You have a list of twenty elements (caves). Inside each element is a list of connecting caves. This means, that if you are in cave #1 (index `0`), it is connected to `CAVES[0]` → `[1, 4, 5]` (note that these numbers inside are zero-based indexes as well!). So, to see what is the index of the second cave connected to the first one you would write `CAVES[0][1]` (you get first element of the list and, then, the second element of the list from inside).

Do exercise #7 to get comfortable with indexing list of lists.

To allow the player to wander, we need to know where their are to begin with. Let us define a new variable called, simply, `player` and assign a random integer between `0` and `19` to it, thus putting the player into a random cave. For this, you will need a `randomint` function from the `random` library. Look at our previous seminar, if you forgot how to use it.

Our player needs to know where they can go, so on each turn we will need to print out the information about which cave the player is in and about the connecting caves (use string formatting to make this look nice). Let this be our first code snippet for the game. The code should look like this

```
# import randint function from the random library

# define CAVES (simply copy-paste the definition)

# create `player` variable and set it to a random number between 0 and 19,
# putting player into a random cave

# print out the list of the connecting caves. Use string formatting.
```

Put your code into exercise #8.

4.3 Wandering around

Now that the player can "see" where they are, let them wander! Use `input()` function to ask for the index of the cave the player wants to go to and store the value in a new variable `move_to`. Remember that `input()` returns a string, so you will need to explicitly convert it to an integer (see Guess-the-Number game, if you forgot how to do it). Now "move" the player to that by assigning the `move_to` value to the `player`. For now, enter only valid numbers, as we will add checks later. To make wandering continuous, put it inside the while loop, so that player wanders until they get to the cave #5 (index `4`). We will have more sensible game-over conditions later on but this will allow you to exit the game without interrupting it from outside. The code should look like this (remember to watch your indentations!).

```
# import randint function from the random library

# define CAVES (simply copy-paste the definition)

# create `player` variable and set it to a random number between 0 and 19,
# putting player into a random cave

# while player is not in the cave #5 (index 4):
    # print out the list of the connecting caves. Use string formatting.
    # get input about the cave the player want to move to, store it in a variable `mov
    # "move" the `player` to the cave they wanted to `move_to`

# print a nice game-over message
```

Put your code into exercise #9.

4.4 Checking whether a value is *in* the list

Right now we trust the player (well, you) to enter the correct index for the cave. Thus, the program will move a player to a new cave even if you enter an index of the cave that is not connected to the current one. Even worse, it will try to move the player to an undefined cave, if you enter an index larger than 19. To check whether an entered index matches one of the connected cave, you need to use `in` conditional statement. The idea is straightforward, if the value is in the list, the statement is `True`, if not, it is `False`.

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

```
## True
```

```
print(4 in x)
```

```
## False
```

Note that you can check *one* value/object at a time. Because a list is also a single object, you will be checking whether it is an element of the other list, not whether all or some of it elements are in it.

```
x = [1, 2, [3, 4]]
# This is False because x has no element [1, 2], only 1, and 2 (separately)
print([1, 2] in x)

# This is True because x has [3, 4] element
```

```
## False
```

```
print([3, 4] in x)
```

```
## True
```

Do exercise #10.

4.5 Checking valid cave index

Now that you know how to check whether a value is in the list, let's use it to validate cave index. Before moving the player, you now need to check whether the entered index is in the list of the connected caves. If this is `True`, you move the player as before. Otherwise, print out an error message, e.g. "Wrong cave index!" without moving a player. Loop ensure that the player will be prompted for the input again.

```
# import randint function from the random library

# define CAVES (simply copy-paste the definition)

# create `player` variable and set it to a random number between 0 and 19,
# putting player into a random cave

# while player is not in the cave #5 (index 4):
    # print out the list of the connecting caves. Use string formatting.
    # get input about the cave the player want to move to, store it in a variable `move_to`
```

```
    # if `move_to` matches on of the connected caves:
      # "move" the `player` to the cave they wanted to `move_to`
    # else:
      # print out an error message

# print a nice game-over message
```

Put your code into exercise #11.

4.6 Checking that string can be converted to an integer

There is another danger with out input: The player is not guaranteed to enter a valid integer! So far we relied on you to behave but in real life, even when people do not deliberately try to break your program, they will occasionally press the wrong button. Thus, we need to check that the *string* that they entered can be converted to an *integer*.

Python string is an object (more on that in a few seminars) with different methods that allow to perform various operations on them. On subset of methods allows you to make a rough check of its content. The one we are interested is str.isdigit() that checks whether all symbols are digits and that the string is not empty (it has at least one symbol). You can follow the link above to check other alternatives such as `str.islower()`, `str.isalpha()`, etc.

Do exercise #12.

4.7 Checking valid integer input

Modify the code that gets the input from the user. First, store the raw string (not converted to an integer) into an intermediate variable, e.g. `move_to_str`. Then, if `move_to_str` is all digits, convert it to an integer, and do the check that it is a valid connected cave index (moving player or printing an error message). However, if `move_to_str` is not all digits, only print the error message. This means you need to have an if-statement inside the if-statement. The outline is below, watch you indentations!

```
# import randint function from the random library

# define CAVES (simply copy-paste the definition)

# create `player` variable and set it to a random number between 0 and 19,
```

```
# putting player into a random cave

# while player is not in the cave #5 (index 4):
    # print out the list of the connecting caves. Use string formatting.
    # get input into a variable `move_to_str`
    # if move_to_str can be converted to an integer:
        # convert move_to_str to integer and store value in move_to
        # if `move_to` matches on of the connected caves:
            # "move" the `player` to the cave they wanted to `move_to`
        # else:
            # print out an error message
    # else:
        # print error message that user must enter a number

# print a nice game-over message
```

Put your code into exercise #13.

4.8 Wrap up

You now have a player in a system of the caves and they can navigate around. Next time, you will learn how to make your code modular by using functions and the game will have more thrills to it once we add bottomless pits and excitable bats.

Cross references

T

- Types

V

- Value types (simple)
- Variables
- Visual Studio Code, a lightweight free open-source editor with strong support for Python. See also installation notes.

4.8.0.1 W{- .unlisted}}

- While loop.