Python for social and experimental psychology

Alexander (Sasha) Pastukhov

2020-12-08

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.

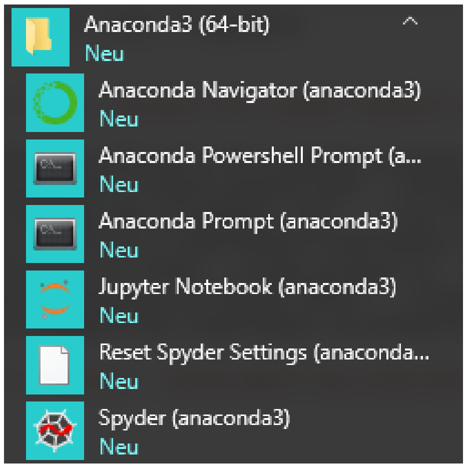
We will learn programming in Python, which is a great language that combines simple and clear syntax with power and ability to tackle almost any problem. The advantage of learning Python, as compared to say Matlab, which is commonly used in neuroscience, is that it allows you do almost anything. In this 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.


Anaconda3 (64-bit)
Neu
Anaconda Navigator (anaconda3)
Neu
Anaconda Powershell Prompt (a...
Neu
Anaconda Prompt (anaconda3)
Neu
Jupyter Notebook (anaconda3)
Neu
Reset Spyder Settings (anaconda...
Neu
Spyder (anaconda3)
Neu

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!).

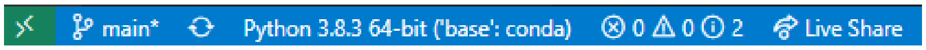
Alternatively, you can start *Jupyter Notebook* directly from the start menu. Please read the online documentation to familiarize yourself with Jupyter Notebook basic interface, e.g. how to create a new cell, run it, etc.

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).

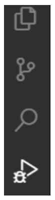
Once the Python extension is activated, you will see which Python interpreter is used (you can have more than one or you may have multiple virtual environments).



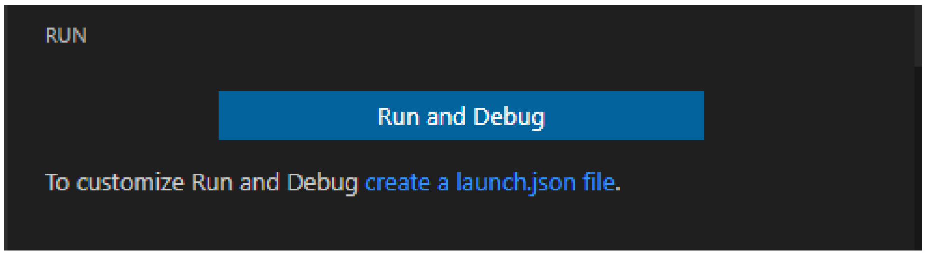
If the selected environment is the wrong one or you are simply not sure, click on it and it will open a drop-down list with all interpreters and environments you have. Consult VS Code online documentation on environments, if you need to change/add/delete environment (the exact settings may change, so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

Debugging in VIsual Studio Code

VS Code gives you comprehensive debugging capabilities (read the tutorial for detailed information about the tools you have). You can run the code in the debugging mode by pressing on debug button on the left toolbar



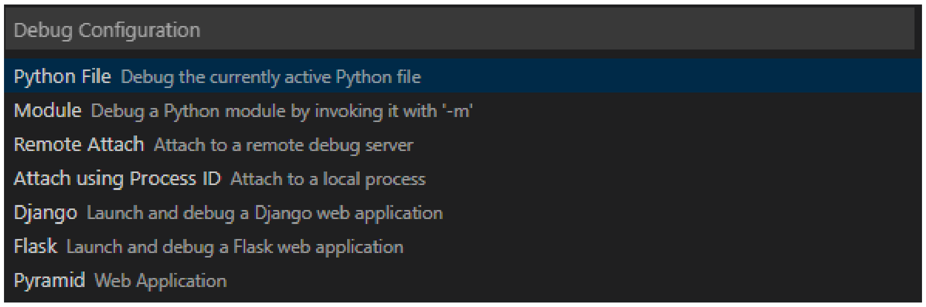
This will open the tab that initially will have the "Run and Debug" button or a suggestion to customize the setup



RUN

To customize Run and Debug create a launch.json file.

Click on "create a launch.json file", which will open a drop-down list at the top of the editor window:



Debug Configuration

Python File Debug the currently active Python file
Module Debug a Python module by invoking it with '-m'
Remote Attach Attach to a remote debug server
Attach using Process ID Attach to a local process
Django Launch and debug a Django web application
Flask Launch and debug a Flask web application
Pyramid Web Application

Pick the first one (Debug the currently active python file) and this will generate a *launch.json* configuration file, which you can close immediately. Now, in an active python file you can press *F5* to start run and debug the program.

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.

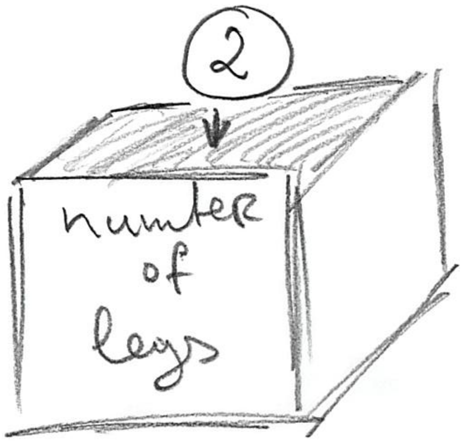
Chapter 1

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
```

```python
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:

```python
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]`:

```python
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 6th 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 they 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 len(CAVES)-1,
# 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 one 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.

Chapter 5

Hunt the Wumpus, part 2

During our previous seminar, we defined a system of interconnected caves, placed a player into a random cave, and allowed them to wander around. Now, we will make the code modular by using functions. Don't forget to download the exercise notebook.

5.1 Functions

In programming, purpose of a function is to isolate certain code that performs a single computation making it testable and reusable. Let us go through the first sentence bit by bit using examples.

Function performs a single computation

I told you that reading code is easy because every action has to be spelled-out for computers in simple and clear way. However, a lot of simple things can be very overwhelming and confusing. Think about the final code for the previous seminar: we had a loop with two conditional statements nested inside the loop and each other. Add a few more of those and you have so many brunches to trace, you will never be quite sure what will happen. This is because our cognition and working memory, which you use to trace all brunches, are limited to just about four items (the official magic number is 7±2 but reading the original paper tells you that this is more like four for most of us).

Thus, a function should perform one computation that is conceptually clear and those purpose should be understood directly from its name or, at most, from a single sentence that describes it[1]. If you need more than once sentence to explain what function does, you should consider splitting the code further. This does not mean that entire description / documentation must fit into a single sentence. The full description can be lengthy, particularly if underlying computation is complex and there are many parameters to consider. However, these are optional details that tell the reader *how* the function is doing its job. Again, they should be able to understand *what* the job is just from the name or from a single sentence. I am repeating myself and stressing it so much because conceptually simple single job functions are a foundation of a clear robust reusable code. And, trust me on this one, future-you will be very grateful that it has to work with easy-to-understand isolated reliable code you wrote.

Function isolates code from the rest of the program

Isolation means that your code is run in a separate scope where the only things that exist are function arguments (limited number of values you pass to it from outside with fixed meaning) and local variables that you define inside the function. You have no access to variables defined in the outside script or to variables defined inside of other functions. Conversely, neither global script nor other function have access to variables and values you compute inside. This means that you only need to study the code *inside* the function to understand how it works. Accordingly, when you write the code it should be *independent* of any global context the function can be used in. Thus, isolation is both practical (no run-time access to variables from outside means fewer chance that things go terribly wrong) and conceptual (no further context is required to understand the code).

Function makes code easier to test

You can build even moderately complex programs only if you can be certain what individual chunks of code are doing under every possible condition. Do they give the correct results? Do the fail clearly raising an error, if the inputs are wrong? Do they use defaults when required? However, testing all chunks together means running extreme number of runs as you need to test all possible combinations of conditions for one chunk given all possible conditions for other chunk, etc. Functions make your life much easier. Because they have a single point of entry, fixed number of parameters, a single return value, and are isolated (see above), you can test them one at a time independent of other functions of the rest of the code. This is called *unit testing* and it is heavy use of automatic unit testing (it is normal to have more code devoted to testing than to the actual

[1]This is similar to scientific writing, where a single paragraph conveys a single idea. So, for me, it helps to first write the idea of the paragraph in a single sentence before writing the paragraph itself. If one sentence is not enough, I need to split the text into more paragraphs.

program) that ensures reliable code for absolute majority of programs and apps that you use.

Function makes code reusable

Sometimes this reason is given as the primary reason to use functions. Turning code into a function means that you can call the function instead of copy-pasting the code. The latter is a terrible idea as it means that you have to maintain the same code at many places (sometimes you might not be even sure in just how many). This is a problem even if the code is extremely simple. Here, we define a *standard* way to compute an initial by taking the first symbol from a string. The code is as simple as it gets.

```python
...
initial = "test"[0]
...
initial_for_file = filename[0]
...
initial_for_website = first_name[0]
...
```

Imagine that you decided to change it and use first *two* symbols. Again, the computation is hardly complicated, use just replace `[0]` with `[:2]`. But you have to do it for *all* the code that does this computation. And you cannot use *Replace All* option because sometimes you might use the first element for some other purposes. And when you edit the code, you are bound to forget about some locations (at least, I do it all the time) making things even less consistent and more confusing. Turning code into a function means you need to modify and test just everything at just one location. Here is the original code implemented via a function.

```python
def generate_initial(full_string):
    """Generates an initial using first symbol.

    Parameters
    ----------
    full_string : str

    Returns
    ----------
    str : single symbol
    """
    return full_string[0]

...
```

```
initial = generate_initial("test")
...
initial_for_file = generate_initial(filename)
...
initial_for_website = generate_initial(first_name)
...
```

and here is the "alternative" initial computation. Note that the code that uses the function *stays the same*

```
def generate_initial(full_string):
    """Generates an initial using first TWO symbols.

    Parameters
    ----------
    full_string : str

    Returns
    ----------
    str : two symbols long
    """
    return full_string[:2]

...
initial = generate_initial("test")
...
initial_for_file = generate_initial(filename)
...
initial_for_website = generate_initial(first_name)
...
```

Thus, turning the code into function is particularly useful when the reused code is complex but it pays off even if computation is as simple and trivial as in example above. With a function you have a single code to worry about and you can be sure that same computation is performed whenever you call the function (not the copy of the code that should be identical but may be not).

Note that I put reusable code as the last reason to use functions. This is because the other three reasons are far more important. Having a conceptually clear isolated and testable code is advantages even if you call this function only once. It still makes code easier to understand and to test and helps you to reduce the complexity by replacing code with its meaning. Take a look at the example below. The first code takes the first symbol but this action (taking the first symbol) does not *mean* anything by itself, it is just a mechanical computation. It is only the original context `initial_for_file = filename[0]` or additional comments that give it its meaning. In contrast, calling a function called *compete_initial* tells you what is happening, as it disambiguates the purpose of the computation. I suspect that future-you is very pro-disambiguation and anti-confusion.

```python
if filename[0] == "A":
    ...

if compute_initial(filename) == "A":
    ...
```

5.2 Defining a function in Python

A function in Python looks like this (note the indentation)

```python
def <function name>(param1, param2, ...):
    some internal computation
    if somecondition:
        return some value
    return some other value
```

The parameters are optional, so is the return value. Thus the minimal function would be

```python
def minimal_function():
    pass # pass means "do nothing"
```

You must defined your function (once!) before calling it (one or more times). Thus, you should create functions *before* the code that uses it (main script or other functions).

```python
def do_something():
    """
    This is a function called "do_something". It actually does nothing.
    It requires no input and returns no value.
    """
    return


def another_function():
    ...
    # We call it in another function.
    do_something()
    ...


# This is a function call (we use this function)
do_something()

# And we use it again!
do_something()
```

Do exercise #1.

You must also keep in mind that redefining a function (or defining a different function that has the same name) overwrites the original definition, so that only the latest version of it is retained and can be used.

Do exercise #2.

Although example in the exercise makes the problem look very obvious, in a large code that spans multiple files and uses various libraries, the issue may not be so straightforward!

5.3 Function arguments

Some function may not need arguments (also called parameters), as they perform a fixed action:

```
def ping():
    """
    Machine that goes "ping!"
    """
    print("ping!")
```

However, typically, you need to pass information to the function, which then affects how the function performs its action. In Python, you simply list arguments within the round brackets after the function name (there are more bells and whistles but we will keep it simple for now). For example, we could write a function that computes and prints person's age given two parameters 1) their birth year, 2) current year:

```
def print_age(birth_year, current_year):
    """
    Prints age given birth year and current year.

    Parameters
    ----------
    birth_year : int
    current_year : int
    """
    print(current_year - birth_year)
```

It is a **very good idea** to give meaningful names to functions, parameters, and variables. The following code will produce exactly the same result but understanding *why* and *what for* it is doing what it is doing would be much harder (so **always** use meaningful names!):

```
def x(a, b):
    print(b - a)
```

When calling a function, you must pass the correct number of parameters and pass them in a *correct order*, another reason for a function arguments to have meaningful names.

Do exercise #3.

When you call the function, the values you *pass* to the function are assigned to the parameters and they are used as *local* variables (more on *local* bit later). However, it does not matter *how* you came up with this values, whether they were in a variable, hard-coded, or returned by another function. If you are using numeric, logical, or string values (*immutable* types), you can assume that any link to the original variable or function that produced it is gone (we'll deal with *mutable* types, like lists, later). Thus, when writing a function or reading its code, you just assume that it has been set to some value during the call and you can ignore the context in which this call was made

```
# hardcoded
print_age(1976, 2020)

# using values from variables
i_was_born = 1976
today_is = 2020
print_age(i_was_born, today_is)

# using value from a function
def get_current_year():
    return 2020

print_age(1976, get_current_year())
```

5.4 Functions' returned value (output)

Your function may perform an action without returning any value to the caller (this is what out `print_age` function was doing). However, you may need to return the value instead. For example, to make things more general, we might want write a new function called `compute_age` that return the age instead of printing it (we can always print ourselves).

```python
def compute_age(birth_year, current_year):
    """
    Computes age given birth year and current year.

    Parameters
    ----------
    birth_year : int
    current_year : int

    Returns
    ----------
    int : age
    """
    return current_year - birth_year
```

Note that even if a function returns the value, it is retained only if it is actually used (stored in a variable, used as a value, etc.). Thus, just calling it will not by itself store the returned value anywhere!

Do exercise #4.

5.5 Scopes (for immutable values)

As we have discussed above, turning code into a function *isolates* it, so makes it run in it own *scope*. In Python, each variable exists in the *scope* it has been defined in. If it was defined in the *global* script, it exists in that *global* scope as a *global* variable. However, it is not accessible (at least not without special effort via a `global` operator) from within a function. Conversely, function's parameters and any variable defined *inside a function*, exists and is accessible only **inside that function**. It is fully invisible for the outside world and cannot be accessed from a global script or from another function. Conversely, any changes you make to the function parameter or local variable have no effect on the outside world (well, almost, *mutable* objects list lists are more complicated, more on that later).

The purpose of scopes is to isolate individual code segments from each other, so that modifying variables within one scope has no effect on all other scopes.

This means that when writing or debugging the code, you do not need to worry about code in other scopes and concentrate only on the code you working on. Because scopes are isolated, they may have *identically named variables* that, however, have no relationship to each other as they exists in their own parallel universes. Thus, if you want to know which value a variable has, you must look only within the scope and ignore all other scopes (even if the names match!).

```python
# this is variable `x` in the global scope
x  = 5

def f1():
  # This is variable `x` in the scope of function f1
  # It has the same name as the global variable but
  # has no relation to it: many people are called Sasha
  # but they are still different people. Whatever you
  # happens to `x` in f1, stays in f1's scope.
  x = 3


def f2(x):
  # This is parameter `x` in the scope of function f2.
  # Again, no relation to other global or local variables.
  # It is a completely separate object, it just happens to
  # have the same name (again, just namesakes)
  print(x)
```

Do exercise #5.

5.6 Create *input_int()*

Let us create the first function called `input_int`. It will take have no arguments (yet) and will return an integer value. This will encapsulate the checks and repeated prompts inside the function, making it easier to maintain the code. It will also make your top-level code cleaner as multiple lines are now replaced with a single call to a function `input_int()`, so you know that in this line you get an integer input from the user. This helps you to concentrate on what is happening ("I am getting an integer input") not how it is happening.

So let us re-implement the code that you created during the last seminar as a function with the only difference is that you `return` the user input instead of using the variable's value directly. I would recommend implementing the code in a separate cell without the function header (`def input():`) and the `return` statements first. Once it works, you can indent it and add the function header. Next, test it by calling the function (e.g. `guess = input_int()` or just `input_int()`), to see that it works reliably, i.e. keeps prompting you until you enter a valid integer.

```
def input_int():
  get user input and store it in a local variable
  while it cannot be converted to an integer:
    remind the player that it must enter an integer
    get user input and store it in a local variable

  return input-as-an-integer
```

Put your code into exercise #6.

5.7 Documenting *input_int()*

Writing a function is only half the job. You need to document it! This may feel excessive but it does not take much time and it is a good habit that makes your code easy to use and reuse. Document your code (a function, or a class, or a module) even if you just trying things out. Remember, "there is nothing more permanent than a temporary solution." Not documenting code is a false economy: a few minutes you save on not documenting it, will translate into dozens of them when you try to understand and debug undocumented code later on.

There are different ways to document the code but we will use NumPy docstring convention. Here is an example of such documented function

```
def generate_initial(full_string):
    """Generates an initial using first symbol.

    Parameters
    ----------
    full_string : str

    Returns
    ----------
    str : single symbol
    """
    return full_string[0]
```

Take the look at the manual and document the function NumPy style. You will only need one-line summary and return value information.

Put your code into exercise #7.

5.8 Adding prompt parameter to *input_int()*

So far the input function you called inside our `input_int()` function either had no prompt or had some fixed prompt that you hard-coded. However, we will use this function for two different actions: moving (the only thing player can do now) and shooting an arrow (we are hunting the Wumpus, after all!). These two actions require two different prompts, so it makes sense to add an *argument* `prompt` to our `input_int()` function.

You assume that this argument is a string that you need to pass on to the `input()` function you are using inside. Thus, you would be able to call your function (almost) the same way as you called `input()`, e.g. `input_int("Please enter the cave index")`, but are guaranteed to get an integer value, as all the check and repeated prompts occur inside of the function.

Put your code into exercise #8.

5.9 Using the function in the code

Now we have a function that makes our code cleaner and easier to understand, so let us use it! Copy-paste your final game code for the previous seminar and alter it to use `input_int()` in place of `input()` + checks + type-conversion. In this modified code, put the function declaration after the import and a constant definition but before the rest of the code.

Put your code into exercise #9.

5.10 Create *input_cave()* function

You implemented a function that get an *integer* input from the player. This is a good first step, as it takes care of all the checks that the value is of the correct type. However, we are not interested in getting an integer per se, we are interested in getting the index of the cave the player wants to move to and this index must be *correct*, as in match the index of accessible caves.

Let us implement a function that does just that. We will call it `input_cave`, it will have a single argument `accesible_caves` (the assumed value is the list of accessible caves), and it will return a integer: the index of the cave the player picked. In the function, you need to print the cave indexes and ask about which cave the player wants to go to until they give a valid answer. Note that you do not need to re-implement the `input_int()` functionality inside, you use that function to get an integer and perform additional checks that integer is `in` the list. Don't forget to document and test it!

Put your code into exercise #10.

Now copy-paste the code from exercise #9 and alter it to use `input_cave` in place of `input_int()` + checks code.

Put your code into exercise #11.

5.11 Create *find_empty_cave()* function

As final modification for today, let us spin-off the code that places the player into a random cave into a separate function. We will call it `find_empty_cave` and, currently, it will have just one parameter `caves_number` (the total number of caves), and it will generate and return a random number between `0` and `caves_number - 1`. Its functionality will be identical to the simple call you are already making in your code, so this may feel unnecessary. However, later we will be placing other objects (bottomless pits, bats, the Wumpus), so we will need a function that can find an empty (unoccupied) cave with all the necessary lack-of-conflict checks. The current function, however limited, will serve as a foundation for our development during the next seminar.

Do not forget to document and test the function. ::: {.infobox .program} Put your code into exercise #12. :::

Now copy-paste the code from exercise #11 and alter it to use `find_empty_cave` function.

Put your code into exercise #13.

Your final code should look roughly as follows and, as you can see, the main script is now slim and easy to follow.

```
# import randint function from the random library

# define CAVES (simply copy-paste the definition)

# define input_int function
# define input_cave function
# define find_empty_cave function

# create `player` variable and put him into an empty (unoccupied) cave

# while player is not in the cave #5 (index 4):
    # get input on which cave the player wants to move to
    # move the player to that cave

# print a nice game-over message
```

Chapter 6

Hunt the Wumpus, part 3

During the last seminar, we used functions to make out code modular. Now, let us add more bells-and-whistles to the game: bottomless pits, crazy bats, and the Wumpus itself. Don't forget to download the exercise notebook.

Before we continue with the game, I would like to introduce the critical difference between *mutable* and *immutable* types.

6.1 Recall, Variables as Boxes (immutable objects)

You may remember the *variable-as-a-box* metaphor. Just to remind you, a variable is "box" with the variable name written on it and a value is stored "inside". When you use this value or assign it to a different variable, you can assume that Python *makes a copy* of it (not really, but this makes it easier to understand) and puts that copy into a different variable "box". When you *replace* value of a variable, what happens is that you take out the old value, destroy it (by throwing into a nearest black hole), create a new one, and put it into the variable "box". When you *change* the value of a variable based on its current state, the same thing happens. You take out the value, create a new value (by adding to the original one or doing some other operation), destroy the old one, and put it back into the variable "box".

This metaphor works nicely and explains why the scopes work the way they do (see the Scopes part in the previous seminar). Each scope has its own set of boxes and whenever you pass information between the scopes, e.g. from a global script to a function, a copy of value is created and put into a new box (i.e., parameter) inside the function. When the function returns the value, that is copied and put in one of the boxes in the global script, etc.

Unfortunately, this is true only for *immutable* objects (values) such as numbers, strings, logical values, but also tuples (see below for what these are). As you could have guessed from the name, this means that there are other *mutable* objects and they behave very differently.

6.2 Variables as post-it stickers (mutable objects)

Mutable objects are lists, dictionaries, and classes. The difference is that *immutable* objects can be thought as fixed in their size. A number takes up that many bytes to store, same goes for a given string (although a different string would require more or fewer bytes). Still, they do not change, they are created and destroyed when unneeded but never truly updated.

Mutable objects can be changed. For example, you can add elements to your list, or remove them, or shuffle them. Same goes for dictionaries. Making such object **immutable** would be computationally inefficient (every time you add a value a long list is destroyed and recreated with just that one additional value), which is why Python simply *updates* the original object. For further computation efficiency, these objects are not copied but *passed by reference*. This means that the variable is no longer a "box" but a "sticker" you put on an object (a list, a dictionary). And, you can put as many stickers on an object as you want **and it still will be the same object!**

What on Earth do I mean? Keeping in mind that a variable is just (one of many) stickers for a mutable object, try figuring out what will be the output below:

```
x = [1, 2, 3]
y = x
y.append(4)
print(x)
```

Do exercise #1.

What? Why? "But I didn't touch `x`, only `y`" I hear you say? That is precisely what I have meant with "stickers on the same object". Since both `x` and `y` are stickers on the *same* object, they are, effectively, synonyms. In that specific situation, once you set `x = y`, it does not matter which variable name you use to change *the* object, they are just two stickers hanging side-by-side on the *same* list. Again, just a reminder, this is **not** what would happen for **immutable** values, like numbers, where things would behaved the way you expect them to behave.

This variable-as-a-sticker, aka "passing value by reference", has very important implications for the function calls, as it breaks your scope without ever giving you a warning. Look at the code below and try figuring out what the output will be.

```
def change_it(y):
    y.append(4)

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

Do exercise #2.

How did we manage to modify a *global* variable from inside the function? Didn't we change the *local* parameter of the function? Yep, that is exactly the problem with passing by reference. Your function parameter is just another sticker on the *same* object, so even though it *looks* like you do not need to worry about global variables (that's why you wrote the function!), you still do. If you are perplexed by this, you are in a good company. This is one of the most unexpected and confusing bits in Python that routinely catches people by surprise. Let us do a few more exercises, before I show you how to solve the scope problem for the mutable objects.

Do exercise #3.

6.3 Tuple: a frozen list

The wise people who created Python were acutely aware of the problem that the *variable-as-a-sticker* creates. Which is why, they added an **immutable** version of a list, called a tuple. It is a "frozen" list of values, which you can loop over, access its items by index, or figure out how many items it has, but you *cannot modify it*. No appending, removing, replacing values, etc. For you this means that a frozen list is a box rather than a sticker and that it behaves just like any other "normal" **immutable** object. You can create a `tuple` by using round brackets.

```
i_am_a_tuple = (1, 2, 3)
```

You can loop over it, *e.g.*

```
i_am_a_tuple = (1, 2, 3)
for number in i_am_a_tuple:
    print(number)
```

```
## 1
## 2
## 3
```

but, as I said, appending will throw a mistake (try this code in a cell)

```
i_am_a_tuple = (1, 2, 3)

# throws AttributeError: 'tuple' object has no attribute 'append'
i_am_a_tuple.append(4)
```

Same goes for trying to change it

```
i_am_a_tuple = (1, 2, 3)

# throws TypeError: 'tuple' object does not support item assignment
i_am_a_tuple[1] = 1
```

This means that when you need to pass a list of values to a function, you should instead pass *a tuple of values* to the function. The function still has a list of values but the link to the original list object is now broken. You can turn a list into a tuple using `tuple()`. Keeping in mind that `tuple()` creates a frozen copy of the list, what will happen below?

```
x = [1, 2, 3]
y = tuple(x)
x.append(4)
print(y)
```

Do exercise #4.

As you probably figured out, when `y = tuple(x)`, Python creates **a copy** of the list values, freezes them (they are immutable now), and puts them into the "y" box. Hence, whatever you do to the original list, has no effect on the immutable "y".

Conversely, you "unfreeze" a tuple by turning it into a list via `list()`. Please note that it creates **a new list**, which has no relation to any other existing list, even if values were originally taken from any of them!

Do exercise #5.

Remember I just said that list() creates a new list? This means that you can use it to create a copy of a list directly, without an intermediate tuple step. You can also achieve the same results by slicing an entire list, e.g. `list(x)`, is the same as `x[:]`.

Do exercise #6.

Here, y = list(x) created a new list (which was a carbon copy of the one with the "x" sticker on it) and the "y" sticker was put on that new list, while the "x" remained hanging on the original.

Confusing? You bet! If you feel overwhelmed by this whole mutable/immutable, tuple/list, copy/reference mess, you are just being a normal human being. I understand the reasons for doing things this way and I am aware of this difference but it still catches me by suprise occasionally!

6.4 Keeping track of occupied caves

So far, we had only the player to keep the track of and we were doing it by storing their location in the `player` variable. However, as we will add more game objects (bottomless pits, bats, the Wumpus), we need to keep the track of who-is-where, so that we don't place them in an already occupied cave. For this, we will keep a *list* of occupied caves (let's call the variable `occupied_caves`). We will initialize it as empty list (you can make an empty list via either `[]` or `list()`) and then append to it (e.g., `list.append(new_value)`). We will pass this list to `find_empty_cave` function as a second argument and we will modify the function to generate an index of the cave *not* in that list.

Let us start with the function, add a second argument to it (call it `caves_taken` or something along these lines) and modify the code so the it keeps randomly generating cave index until it is *not* in the `caves_taken` list. Remember to document the code and test the code by passing a list of hard-coded values to see that it never returns value from the list (e.g., `find_empty_cave(len(CAVES), [2, 5, 7])`).

Put your code into exercise #7.

Now we need to modify the main script to take advantage of the update `find_empty_cave()`. Here is the outline

```
# import randint function from the random library

# define CAVES

# define input_int function
# define input_cave function
-> # define updated find_empty_cave function

-> # create `occupied_caves` variable and initialize it to an empty list
-> # create `player` variable and put him into an empty (unoccupied) cave (use `occupied_caves` u
-> # append player location to the `occupied_caves` list for future use
```

```
# while player is not in the cave #5 (index 4):
    # get input on which cave the player wants to move to
    # move the player to that cave

# print a nice game-over message
```

Put your code into exercise #8.

Now that we have the scaffolding in place, let us add bottomless pits. But before that, you need to learn about for loops and range.

6.5 For loop

The for loop iterates over sequence of items. Generally speaking, for loop loops over items that are in *iterable* containers or yielded by *generators*. We won't go into exact details of what these are and how their are implemented. For now, you just need to know that there are two kinds of things a for loop can iterate over and that they differ in whether they produce a finite number of elements to loop over (iterable containers) or a potentially endless number of elements to iterate over (generators). An example of the former is a list (an iterable container that holds a finite number of items), an example of the latter is a `range()` function, which we will discuss separately below.

To iterate over items in the list, you simply write

```
for item in list:
  ...
  do something with an item or just do something
  ...
```

For example, you can print every item of the list one-by-one. The loop will be executed three times (there are three items in the list) and on each iteration the `item` variable will take the value of the corresponding element: `"A"` on the first iteration, `"B"` on the second, `"C"` on the third.

```
for item in ["A", "C", "K"]:
    print("Item is %s"%(item))
```

```
## Item is A
## Item is C
## Item is K
```

Do exercise #9.

6.6 range()

Sometimes you need to repeat an action several times (e.g., in our code below, we will need to add two bottomless pits) or iterate over sequence of numbers. You can write down such list by hand, e.g. `[0, 1, 2]`, but an easier way is to use range(start, stop, step) function that generates a sequence of numbers from `start` until-but-not-including `stop` with a given `step`. If you omit the `step`, it is assumed to be `1`. You can also pass just one value that is interpreted as `stop` with `start=0` and `step=1`. Thus, `range(3)` is the same as `range(0, 3)` and `range(0, 3, 1)`.

Important note, `range()` function does not produce a list of values but a *generator* that yields values one at a time. You can see it by calling this function in isolation

```
range(3)
```

```
## range(0, 3)
```

The generators are functions (or objects) that yield items upon request, which makes them "lazy" but more memory efficient. E.g., if you want to iterate over 1,000,000 numbers, you do not need to generate a 1,000,000 number long list (takes memory), you just need to get numbers from that sequence one a time. You can convert a generator into a list via `list()` function

```
list(range(3))
```

```
## [0, 1, 2]
```

The reason for distinction between generators and iterable containers like lists, is that generator sequences can be unlimited. For example, you can have a generator that yields a new random number upon request. It will never run out of items to yield and trying to convert it to a list would be a bad idea as an infinitely long list requires infinite amount of memory!

Do exercise #10.

6.7 Placing bottomless pits

Now we can add bottomless pits to the game. The idea is simple, we place two of those in random caves, so when the player wanders into a cave with a bottomless pit, they fall down and die (game over). We will, however, warn the player, that their current cave is next to a bottomless pit, without telling them which cave it is in specifically.

First thing first, let us add them. For this, we will create a new constant `NUMBER_OF_BOTTOMLESS_PITS` (I suggest that we set it to `2` but you can have more or fewer of them) and a new variable (`bottomless_pits`) that will hold a list of indexes of caves with the bottomless pits in them. Add bottomless pits to using using a for loop: On each iteration get an index of an empty cave (via `find_empty_cave` functionm think about its parameters), append this index to both 1) `bottomless_pits` and 2) `occupied_caves` variables, so that you 1) know where bottomless pits are and 2) know which caves are occupied. Here is the code outline for the initialization part (do not copy paste the main loop just yet). See if numbers makes sense (number of caves is what you expected them to be, value are within the range, there are no duplicates, etc.)

```
# import randint function from the random library

# define CAVES
-> # define NUMBER_OF_BOTTOMLESS_PITS

# define input_int function
# define input_cave function
# define find_empty_cave function

# create `occupied_caves` variable and initialize it to an empty list
# create `player` variable and put him into an empty (unoccupied) cave (use `occupied_
# append player location to the `occupied_caves` list for future use

-> # create `bottomless_pits` variable and initialize it to an empty list
-> # use for loop and range function to repeat the for loop NUMBER_OF_BOTTOMLESS_PITS
-> #     generate a new location for the bottomless pit via find_empty_cave() function
-> #     append this location to `occupied_caves` variable
-> #     append this location to `bottomless_pits` variable


-> # print out both player and bottomless_pits variables for diagnostics
```

Put your code into exercise #11.

6.8 Falling into a bottomless pit

Now we will add one of the ways for the game to be over: the player falls into a bottomless pit. For this, we just need to check whether player is currently in a cave that has a bottomless pit in on every turn. If that is the case, player's cave is indeed in the bottomless pits list, print a sad game over message and `break` out of the loop. In addition, let us modify the `while` loop condition to `while True:`, so that the only way to end the game is to fall into the pit (not exactly fair to the player, but we'll fix that later). The outline of the updated code is as follows:

```
# import randint function from the random library

# define CAVES
# define NUMBER_OF_BOTTOMLESS_PITS

# define input_int function
# define input_cave function
# define find_empty_cave function

# create `occupied_caves` variable and initialize it to an empty list
# create `player` variable and put him into an empty (unoccupied) cave (use `occupied_caves` wher
# append player location to the `occupied_caves` list for future use

# create `bottomless_pits` variable and initialize it to an empty list
# use for loop and range function to repeat the for loop NUMBER_OF_BOTTOMLESS_PITS times
#     generate a new location for the bottomless pit via find_empty_cave() function
#     append this location to `occupied_caves` variable
#     append this location to `bottomless_pits` variable

-> # print out both player and bottomless_pits variables for diagnostics

-> # while True:
    # get input on which cave the player wants to move to
    # move the player to that cave

    -> # if players in a cave with bottomless pit:
    ->      # print out sad game-over message
    ->      # break out of the loop
```

Put your code into exercise #12.

6.9 Warning about a bottomless pit

We need to give the player a chance to avoid the fate of falling into a bottomless pit but warning him that one (or two) are nearby. To this end, we need to print additional information before they decide to make their move. In a for loop, iterate over the connected caves and every time cave has a bottomless pit in it, print “You feel a breeze!”. This informs the player that the cave is nearby and the number of messages tells them just how many bottomless pits are nearby.

Put your code into exercise #13.

6.10 Placing bats

We need more thrills! Let us add bats to the fray. They live in caves, the player can hear them, if they are in a connect cave (“You hear flapping!”), but if the player inadvertently enters the cave with bats, they carry the player to a *random* cave.

Placing the bats is analogous to placing bottomless pits. You need a constant that determines the number of bat colonies (e.g., `NUMBER_OF_BATS` and set it `2` or some other number you prefer), a variable that holds a list with indexes of caves with bats (e.g., `bats`), and you need to pick random empty caves and store them in `bats` in exactly the same way you did it with bottomless pits. Print out location of bats for diagnostic purposes.

Put your code into exercise #14.

6.11 Warned about bats

In the same loop over connected caves that you use to warn the player about bottomless pits, add another check that prints out `"You hear flapping!"` every time the connected cave have bats in it.

Put your code into exercise #15.

6.12 Transported by bats to a random cave

If the player is in the cave with bats, they transport them to a *random* cave, irrespective of whether the cave is occupied or not. Thus, bats can carry the player to a cave:

1. with another bat colony and it will transport the player again.
2. with a bottomless pit and the player will fall into it.
3. later on, to the cave with the Wumpus (the player may not survive that one).

Think about:

1. *when* you check for bats presence (before or after checking for a bottomless pit?),
2. do you check once (using `if`) or one-or-more times (using `while`)

Put your code into exercise #16.

Next time, we will finish the game by adding the Wumpus and the arrows.

Chapter 7

Hunt the Wumpus, part 4

Our game is almost complete, we only need to add the Wumpus and arm the player with crooked arrows! Grab the exercise notebook and let get busy.

7.1 Adding Wumpus.

By now you have added a player (single, location stored as in integer), bottomless pits (plural, locations stored in a list), and bats (plural as well). Add Wumpus!

1. Create a new variable (`wumpus`?) and place Wumpus in an occupied cave. Print out location of Wumpus for debugging purposes.
2. Warn about Wumpus in the same code that warns about pits and bats. Canonical warning text is `"You smell a Wumpus!"`.
3. Check if player is in the same cave as Wumpus. If that is the case, game is over, as the player is eaten by a hungry Wumpus. This is similar to *game-over* due to falling into a bottomless pit. Think about whether the check should before or after check for bats.

Put your code into exercise #1.

7.2 Giving player a chance.

Let us give player a chance. As they enter the cave with the Wumpus, they startle it. Then, Wumpus either runs away to a random adjacent cave (new) or stays put and eats the player. First, create a new constant that defines a probability that Wumpus runs away, e.g. `P_WUMPUS_SCARED`. In implementations I've found, it is typically 0.25, but use any value you feel is reasonable.

Thus, if the player is in the cave with Wumpus, draw a random number between 0 and 1. The function you are looking for is random() and it is part of random library, so the call is `random.random()`.If that number is smaller than probability that the Wumpus is scared, move it to a random adjacent cave (bats ignore Wumpus and it clings to the ceiling of the caves, so bottomless pits are not a problem for it). A useful function is choice(), again, part of random library. Otherwise, if Wumpus was not scared off, the player is eaten and game is over (the only outcome in exercise #1).

Put your code into exercise #2.

7.3 Flight of a crooked arrow

Our player is armed with *crooked* arrows that can fly through caves. The rules for its flight are the following:

- The player decide in which cave it shoots an arrow and how far the arrow flies (from 1 up to 5 caves).
- Every time the arrow needs to flight into a next cave, that cave is picked randomly from adjacent caves *excluding* the cave it came from (so, the arrow cannot make a 180° turn and there are only two out of three caves available for choosing).
- If the arrow flies into a cave with Wumpus, it is defeated and the game is won.
- If the arrow flies into a cave with the player, then they committed unintentional suicide and the game is lost.
- If the arrow reached it last cave (based on how far the player wanted to shoot) and the cave is empty, it drops down on the floor.
- Bats or bottomless pits have no effect on the arrow.

The total number of arrows the player has at the beginning should be defined in `ARROWS_NUMBER` constant (e.g., `5`).

To keep track of the arrow, you will need following variables:

- `arrow`: current location of the arrow.
- `arrow_previous_cave`: index of the cave the arrow came from, so that you know where it cannot flight back.
- `shooting_distance`: remaining distance to travel.
- `remaining_arrows`: number of remaining arrows (set to `ARROWS_NUMBER` when the game starts).

7.4 Random cave but no 180° turn

You need to program a function (call it `next_arrow_cave()`) that picks a random cave but not the previous cave the arrow had been in. It should have two parameters:

- `connected_caves`: a list of connected caves.
- `previous_cave`: cave from which the arrow came from.

First, debug the code in a separate cell. Assume that `connecting_caves = CAVES[1]` (so, arrow is currently in cave 1) and `previous_cave = 0` (arrow came from cave 0). Write the code that will pick one of the remaining caves randomly (in this case, either `2` or `7`). Once the code works, turn it into a function that returns the next cave for an arrow. Document the function. Test it with for other combinations of connected and previous caves.

Put your code into exercise #3.

7.5 Going distance

Now that you have a function that flies to the next random cave, implement flying using a for loop. An arrow should fly through `shooting_distance` caves (set it to `5`, maximal distance, by hand for testing). The *first* cave is given (it will be picked by the player), so set `arrow` to `1` and `arrow_previous_cave` to `0` (player is in the cave `0` and shot the arrow into cave `1`). For debugging purposes, print out location of the arrow on each iteration. Test the code by changing `shooting_distance`. In particular, set it to `1`. The arrow should "fall down" already in cave `1`.

Put your code into exercise #4.

7.6 Hitting a target

Implement check for hitting the Wumpus or the player in the loop. Should the check be before or after the arrow flies to the next random cave? In both cases, write an appropriate "game over" message, set variable `gameover` to `True` (we will add to the main code later), and break out of the loop. Test the code by placing Wumpus by hand into the cave the player is shooting at or the next one. Run code multiple times to check that it works.

Put your code into exercise #5.

7.7 Almost where

We are almost where but before we can start putting the code together we need a few more things. To better understand what all the functions are for, take a look at the overall program we are trying to make. You have most parts: placing player and game objects, moving player, checking for bats, bottomless pits, and wumpus. But we still need to implement asking player whether they want to shoot or move, asking for shooting distance, and for cave the player wants to shoot at.

```
import random

define CAVES
define other constants

define functions

place player, bottomless pits, bats, and wumpus
set number of remaining arrows to ARROWS_NUMBER

set gameover to False
while not gameover:
    while player wants to shoot and has arrows:
        ask about cave that player want to shoot at
        ask how the far the arrow should fly

        fly arrow through caves in for loop:
            if hit wumpus -> congrats game over message, gameover=True, and break out o
            if hit player -> oops game over message, gameover=True, and break out of th
        decrease number of remaining arrows

    check if game is over, break out of the loop, if that is the case

    ask player about the cave he want to go to and move player

    check for bats, move player to a random cave, while they are in the cave with bats

    check for bottomless pits (player dies, set gameover to True, break out of the loop

    if player is in the same cave as wumpus:
        if wumpus is scared
            move wumpus to a random cave
        otherwise
            player is dead, set gameover to True, break out of the loop
```

7.8 Move or shoot?

Previously, the player could only move, so we just asked for the next cave number. Now, on each turn, the player will have a choice of shooting an arrow or moving. Implement a function `input_shoot_or_move()` that has no parameters and returns `"s"` for shooting o `"m"` for moving. Inside, ask the player about their choice until they pick one of two options. Conceptually, this is very similar to your other input functions (`input_int()` and `input_cave()`) that repeatedly request input until a valid one is given. Test and document!

Put your code into exercise #6.

7.9 How far?

Implement `input_distance()` function that has no parameters and returns the desired shooting distance between `1` and `5`. Inside, repeatedly ask for an integer number input on how far the arrow should travel until valid input is given. This is very similar to your other input functions. Test and document.

Put your code into exercise #7.

7.10 input_cave prompt

Add `prompt` parameter to the `input_cave()` function you have previously. Now we can ask either about moving to or shooting at the cave, hence, the need for the prompt in place of a hard-coded message.

Put your code into exercise #8.

7.11 Putting it all together

Here is the pseudocode again. Take a look to better understand how the new bits get integrated into the old code. By now, you should have following constants (you can have other values):

- `CAVES`
- `NUMBER_OF_BATS` = 2
- `NUMBER_OF_BOTTOMLESS_PITS` = 2
- `P_WUMPUS_SCARED` = 0.25
- `ARROWS_NUMBER` = 5

Following functions:

- `find_empty_cave(total_caves, caves_taken)`, returns an integer cave index
- `input_int(prompt)`, returns an integer
- `input_cave(prompt, connected_cave)`, returns an integer cave index
- `input_shoot_or_move()`, returns `"s"` for shoot and `"m"` for move.
- `input_distance()`, returns an integer between 1 and 5
- `next_arrow_cave(connected_caves, previous_cave)`, return an integer cave index

Following variables:

- `player` : cave index
- `bottomless_pit`: list of cave indexes
- `bats`: list of cave indexes
- `wumpus`: cave index
- `remaining_arrows`: integer number of remaining arrows
- `gameover`: `False` initially, until something good (defeated wumpus) or bad (fell into a bottomless pit, got eaten by wumpus) happens.

Service/temporary variables:

- `occupied_caves`: list of cave indexes
- `arrow`: location of an arrow
- `shooting_distance`: number of caves the arrow should fly through.

```
import random

define CAVES
define other constants

define functions

place player, bottomless pits, bats, and wumpus
set number of remaining arrows to ARROWS_NUMBER

set gameover to False
while not gameover:
    while player wants to shoot (input_shoot_or_move function) and has arrows (remainin
        ask about cave that player want to shoot at (input_cave function), store answer
        ask how the far the arrow should fly (input_distance function), store answer in

        fly arrow through caves in for loop (shooting_distance caves):
            if hit wumpus -> congrats game over message, gameover=True, and break out o
            if hit player -> oops game over message, gameover=True, and break out of th
            move arrow to the next random cave (next_arrow_cave function and arrow variable)
        decrease number of remaining arrows (remaining_arrows variable)

    check if game is over, break out of the loop, if that is the case

    ask player about the cave he want to go to and move player (input_cave function)

    while player is in the cave with bats:
        move player to a random cave

    check for bottomless pits (player dies, set gameover to True, break out of the loop)

    if player is in the same cave as wumpus:
        if wumpus is scared
            move wumpus to a random cave
        otherwise
            player is dead, set gameover to True, break out of the loop
```

Put your code into exercise #9.

7.12 Wrap up

Well done! Next time, we will put *video* into *video game*!

Chapter 8

Gettings start with PsychoPy

Before we program our first game using PsychoPy, we need to spend some time figuring out its basics. It is not the most suitable library for writing games, for that you might want to use Python Arcade or PyGame). However, it is (IMHO) the best Python library for developing psychophysical experiments, which is why we will use it in our course.

For this and following projects, we won't use Jupyter Notebooks but will develop a Python program using IDE environment of your choice (I would recommend Visual Studio Code). You still could and should use Jupyter for playing with and testing small code snippets, though. I've added a section on setting up debugging in VS Code in Getting Started, take a look once you are ready to run the code.

From now on, create a separate subfolder for each seminar (e.g. *Seminar 08* for today) and create a separate file (or files, later on) for each exercise[1] (e.g., *exercise01.py*, *exercise02.py*, etc.). This is not the most efficient implementation of a version control and will certainly clutter the folder. But it would allow me to see your solutions on every step, which will make it easier for me to write comments. For submitting the assignment, just zip the folder and submit the zip-file.

8.1 Minimal PsychoPy code

In the subfolder for the current seminar, create file *exercise01.py* (I would recommend using lead zero in *01*, as it will ensure correct file sorting once we get

[1]You can "Save as…" the previous exercise to avoid copy-pasting things by hand.

to exercise 10, 11, etc.). Copy-paste the following code:

```
"""
A minimal PsychoPy code.
"""

# this imports two modules from psychopy
# visual has all the visual stimuli, including the Window class
# that we need to create a program window
# event has function for working with mouse and keyboard
from psychopy import visual, event

# creating a 800 x 600 window
win = visual.Window(size=(800, 600))

# waiting for any key press
event.waitKeys()

# closing the window
win.close()
```

Run it to check that PsychoPy work. You should get a gray window with *PsychoPy* title. Press any key (click on the window, if you switched to another one, so that it registers the key press) and it should close. Not very exciting but does show that everything works as it should.

Put your code into *exercise01.py*.

Let me explain what we are doing here line-by-line.

- `from psychopy import visual, event` here we import two (out of many) PsychoPy modules that give us *visual* stimuli and main program window plus ability to process *events*, such as keyboard presses or mouse activity.
- `win = visual.Window(size=(800, 600))` we create a new PsychoPy Window *object* (you will learn about classes and objects soon) and define its `size` as 800 by 600 pixels (you can change that and see how the window also changes its size).
- `event.waitKeys()` function waitKeys() waits for a press of a keyboard key. As we didn't specify which keys we are interested in, *any* key will do. Later on, you will learn how to make it wait for specific keys.
- `win.close()` this calls a *method* `close` (function that belongs to an object, again, you'll learn about them later) that tells window `win` to close itself.

8.2 Adding main loop

Currently, nothing really happens in the code, so let us add the main loop. The loop goes between opening and closing the window:

```
importing libraries
opening the window

--> our main loop <--

closing the window
```

For this, you need to create a new variable `gameover` and set it too `False` (just like we did it in *Hunt the Wumpus* game) and run the loop for as long as the game is **not** over. Inside the loop, use function event.getKeys() to check whether *escape* button was pressed (for this, you need to pass `keyList=['escape']`). The function returns a list of keys, if any of them were pressed in the meantime or an empty list, if no keys from the list were pressed. Accordingly, you need to check whether the return value as an empty list. There are two ways to do this. First, you can check whether length of the list is larger than zero (so, it has elements) using function len(). Alternatively, an empty list evaluates to `False` when converted to a logical value either explicitly (via `bool()` type conversion) or when evaluated inside of the condition in an `if` or `while` statement:

```
x = []
if x:
  print("List is not empty")
else:
  print("List is definitely empty")
```

```
## List is definitely empty
```

If list is *not* empty, you should change `gameover` to `True`. Think, how can you do it *without* an `if` statement, computing the logical value directly?

Put your code into *exercise02.py*.

8.3 Adding text message

Although we are now running a nice loop, we still have only a boring gray window to look at. Let us create a text stimulus, which would say "Press escape to exit" and display it during the loop. For this we will use visual.TextStim class from PsychoPy library.

First, you need to create the `press_escape_text` object (instance of the `TextStim` class) before the main loop. There are quite a few parameters that you can play with but minimally, you need to pass the program window (our `win` variable) and the actual text you want to display (`"Press escape to exit"`). For all other settings PsychoPy will use its defaults (default font family, color and size, placed right the screen center).

```
press_escape_text = visual.TextStim(win, "Press escape to exit")
```

To show the visuals in PsychoPy, you first *draw* each element by calling its `draw()` method (thus, in our case, `press_escape_text.draw()`) and then put the "drawing" on the screen by *flipping* we window (`win.flip()` method). These two calls should go inside the main loop either before (my preference) or after the keyboard check.

```
importing libraries
opening the window

--> create press_escape_text here <--

gameover = False
while not gameover:
    --> draw press_escape_text <--
    --> flip the window  <--
    check keyboard for escape button press

close the window
```

Now, you should have a nice, although static, message that tells you how you can exit the game. Check out the manual page for visual.TextStim and try changing it by passing additional parameters to the object constructor call. For example you can change its `color`, whether text is `bold` and/or `italic`, how it is aligned, etc. However, if you want to change *where* the text is displayed, read on below.

Put your code into *exercise03.py*.

8.4 Adding a square and placing it *not* at the center of the window

Now, let us figure out how create and move visuals to an arbitrary location on the screen. In principle, this is very straightforward as every visual stimulus (including `TextStim` we used just above) has `pos` property that specifies (you guessed it!) its position. However, to make your life easier, PsychoPy first complicates it by having **five** different units systems.

Before we start exploring the units, let us create a simple white square. The visual class we need is visual.Rect. Just like the `TextStim` above, it requires `win` variable (so it knows which window it belongs to), `width` (defaults to 0.5 of that mysterious units), `height` (also defaults to 0.5), `pos` (defaults to (0,0)), `lineColor` (defaults to `white`) and `fillColor` (defaults to `None`). Thus, to get a "standard" white square with size of `(0.5, 0.5)` units at `(0, 0)` location you only need pass the `win` variable: `white_square = visual.Rect(win)`. You draw the square just like you drew the text stimulus, via its `draw()` method. Create the code, run it to see a very white square, and read on.

```
importing libraries
opening the window

--> create white_square here <--

gameover = False
while not gameover:
    --> draw white_square and flip the window here <--
    check keyboard for escape button press

close the window
```

Put your code into *exercise04.py*.

What did you say, your square was not really a square? Well, I told you, **five** units systems!

8.5 Five units systems

8.5.1 Height units

With height units everything is specified in the units of window height. The center of the window is at `(0,0)` and the window goes vertically from `-0.5` to `0.5`. However, its horizontal limits depend on the aspect ratio. For our 800×600 window, it will go from -0.666 to 0.666 (the window is 1.3333 window heights wide). For a 600×800 window from -0.375 to 0.375 (the window is 0.75 window heights wide), for a square window 600×600 from -0.5 to 0.5 (again, in all these cases it goes from -0.5 to 0.5 vertically). This means that the actual on-screen distance for the units is the same for both axes. So that a square of `size=(0.5, 0.5)` is actually a square (it spans the same distance vertically and horizontally). Thus, it makes *sizing* objects easier but *placing them on horizontal axis correctly* harder (as you need to know the aspect ratio).

Modify your code by specifying the unit system when you create the window: `win = visual.Window(..., units="height")`. Play with your code by specifying position of the square when you create it. You just need to pass an extra parameter `pos=(<x>, <y>)`. Which was is up, when y is below or above zero?

Put your code into *exercise05.py*.

Unfortunately, unlike x-axis, the y-axis can go both ways. For PsychoPy y-axis points up (so negative values move the square down and positive up). However, if you would use an Eyelink eye tracker to record where participants looked *on the screen*, it assumes that y-axis starts at the top of the screen and points down (which could be very confusing, if you forget about this when overlaying gaze data on the image you used and wondering what on Earth the participants were doing).

Now, modify the size of the square (and turn it into a non-square rectangle) by passing `width=<some-width-value>` and `height=<some-height-value>`.

Put your code into *exercise06.py*.

8.5.2 Normalized units

These units are default ones and assume that the window goes from -1 to 1 both along x- and x-axis. Again, (0,0) is the center of the screen but the top-left corner is (-1, -1) whereas the bottom-right is (1, 1). This makes *placing* your objects easier but *sizing* them harder (you need to know the aspect ratio to ensure that a square is a square).

Modify your code, so that it uses `"norm"` units when you create the window and size the `Rect` stimulus, so it does look like a square.

Put your code into *exercise07.py*.

8.5.3 Pixels on screen

In this case, the window center is still at `(0,0)` but it goes from `-<width-in-pixels>/2` to `<width-in-pixels>/2` horizontally (from -400 to 400 in our case) and `-<height-in-pixels>/2` to `<height-in-pixels>/2` vertically (from -300 to 300). These units could be more intuitive when you are working with a fixed sized window, as the span is the same along the both axes (like for the height units). However, they spell trouble if your window size was changed or you are using a full screen window on some monitor with an unknown resolution.

Modify your code to use `"pix"` units and briefly test sizing and placing your square within the window.

Put your code into *exercise08.py*.

8.5.4 Degrees of visual angle

Unlike the three units above, these require you knowing a physical size of the screen, its resolution, and your viewing distance (how far your eyes are away from the screen). They are *the* measurement units used in visual psychophysics as they describe stimulus size as it appears on the retina (see Wikipedia for details). Thus, these are the units you want to use when running an actual experiment in the lab but for our purposes we will stick to one of the three units systems above.

8.5.5 Centimeters on screen

Here, you would need know the physical size of your screen and its resolution. These are fairly exotic units for very specific situations[2].

8.6 Make your square jump!

So far, we fixed the location of the square when we created it. However, you can move it at any time by assigning a new `(<x>, <y>)` coordinates to its `pos` property. *E.g.*, `white_square.pos = (-0.1, 0.2)`. Let us experiment by moving the square to a random location on every iteration of the loop (this could cause a lot of flashing, so if you have a photosensitive epilepsy that can be triggered by flashing lights, you probably should do it just once before the loop). Use the units of your choice (I would recommend `"norm"`) and generate a new position using random.uniform(a, b) function, that generates a random value within $a..b$ range. You need to generate two values (one for x, one for y) and your range is the same for `"norm"` units (from -1 to 1) but is different (and depends on the aspect ratio) for `"height"` units.

```
importing libraries, now also the random library
open the window
create white_square here

gameover = False
while not gameover:
    --> move the square to a random position <--
    draw white_square and flip the window here
    check keyboard for escape button press

close the window
```

Put your code into *exercise09.py*.

[2]so specific that I can't think of one, to be honest.

8.7 Make the square jump on your command!

This was very flashy, so let us make the square jump only when you press *Space* button. For this, we need to expand the code that processes keyboard input. So far, we restricted it to just "escape" button and checked whether any (hence, "escape") button was pressed. In my case, the code looks like that

```
gameover = event.getKeys(keyList=['escape'])
```

But I could have written it as

```
keys_pressed = event.getKeys(keyList=['escape'])
if keys_pressed is not None:
    game_over = True
```

Let us use the *second* version, where we explicitly store the output of `event.getKeys()` function in `keys_pressed`. First, you need to add `"space"` to the `keyList` parameter. Second, because `event.getKeys()` function returns a **list** of keys that were pressed, we need to loop over that list (which in most cases will contain no or just one element) and use conditional statements inside that loop to make the square jump or to exit the program.

Use for `for` loop to loop over the elements of the list. Note that no iterations will occur if the list is empty, you can test this in a Jupyter cell:

```
for _ in []:
    print("No one will ever see me...")
```

When you loop over `keys_pressed`, your current loop variable value will be the string with the name of the button pressed (so, either `"escape"` or `"space"`). Now, you need to use conditional statements, so that the square jumps if the key was `"space"` and game is over if the key was `"escape"`. You code should look roughly like that

```
importing libraries, now also the random library
open the window
create white_square

gameover = False
while not gameover:
    draw white_square and flip the window here

    # ----- New code -----
    keys_pressed = event.getKeys(keyList=['escape', 'space'])
    loop over keys_pressed:
        if participant pressed space key:
            move the square to a random location
        elif participant pressed escape key:
            set gameover to False
    # ----- End of new code -----

close the window
```

Put your code into *exercise10.py*.

8.8 I like to move it, move it!

Let us exert more control over our rectangle by moving it around using arrow buttons (`"up"`, `"down"`, `"left"`, and `"right"` in PsychoPy). Add these keys to the `keyList` parameter of the `event.getKeys()` call and add more conditional statements when processing the pressed key. In PsychoPy you can change position by adding to it via `+=` or `-=` operations (other operations are also supported, see manual). Thus, you can move your square to the right by 0.1 units (whatever they are) by writing `white_square.pos += (0.1, 0)`. Please note that you **cannot** write `white_square.pos = white_square.pos + (0.1, 0)`!

Expand your code, so you move the rectangle around by 0.05 units in the direction of the key pressed.

Put your code into *exercise11.py*.

When we continue, we will expand on this to build a Memory game. In the meantime, experiment with stimuli (you can have a circle or a line rather than a square). Try showing more than one stimulus (*e.g.*, add back the “press escape to exit” message), etc.

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.

8.8.0.1 W{- .unlisted}}

- While loop.