Python for social and experimental psychology

Alexander (Sasha) Pastukhov

2021-04-14

Contents

Introduction

About the material

About the seminar

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

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

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

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

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

Note on exercises

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

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

Why Python?

The ultimate goal of this seminar is to teach you how to create an experiment for psychology research. There are many ways to achieve this end. You can use drag-and-drop systems either commercial like Presentation, Experiment Builder or free like PsychoPy Bulder interface. They have a much shallower learning curve, so you can start creating and running your experiments faster. However, the simplicity of their use has a price: They are fairly limited in which stimuli you can use and how you can control the presentation schedule, conditions,

feedback, etc. Typically, they allow you to extend them by programming the desired behavior but you do need to know how to program to do this. Thus, I think that while these systems, in particular PsychoPy, are great tools to quickly bang a simple experiment together, they are most useful if you understand how they create the underlying code and how you would program it yourself. Then, you will not be limited by the software, as you know you can program something the default drag-and-drop won't allow, but you can always opt in, if drag-and-drop is sufficient but faster. At the end, it is about having options and creative freedom to program an experiment that will answer your research question, not an experiment that your software allows you to program.

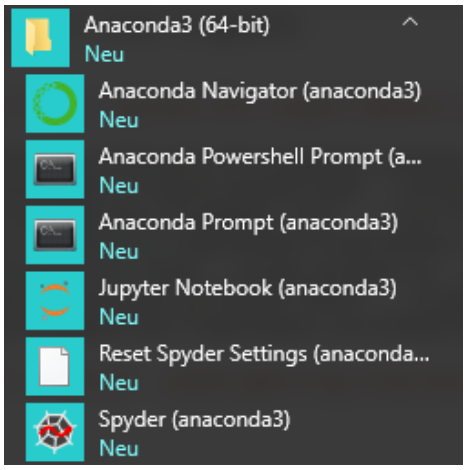
We will learn programming in Python, which is a great language that combines simple and clear syntax with power and ability to tackle almost any problem. The advantage of learning Python, as compared to say Matlab, which is commonly used in neuroscience, is that it allows you do almost anything. In this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

Getting Started

Installing Anaconda environment

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

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



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

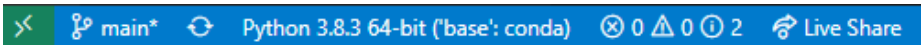
Alternatively, you can start *Jupyter Notebook* directly from the start menu. Please read the online documentation to familiarize yourself with Jupyter Notebook basic interface, e.g. how to create a new cell, run it, etc.

Installing Visual Studio Code

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

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

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

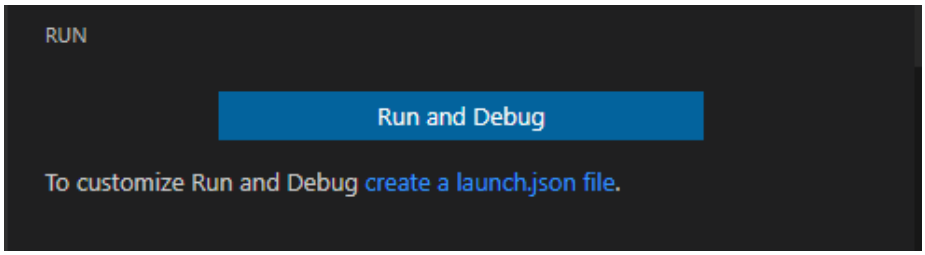


If the selected environment is the wrong one or you are simply not sure, click on it and it will open a drop-down list with all interpreters and environments you have. Consult VS Code online documentation on environments, if you need to change/add/delete environment (the exact settings may change, so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

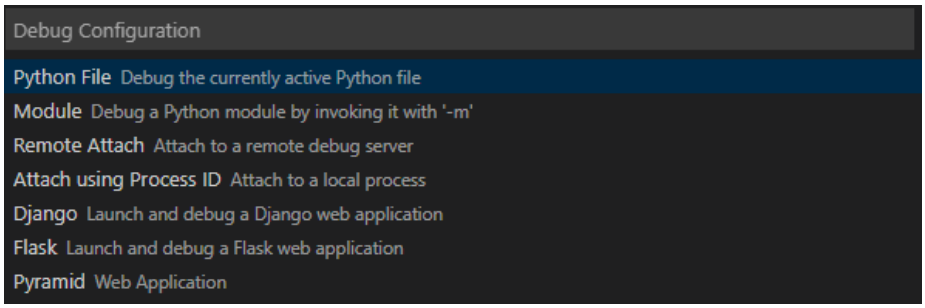
Debugging in VIsual Studio Code

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



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



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.

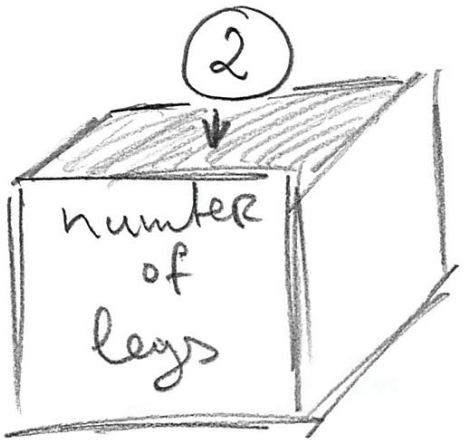
Chapter 1

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
```

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

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

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

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

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

2.5 Conditions and comparisons (exercises 3-8)

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

For the comparison, you can use the following operators:

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

Go to exercise #3 to solve some comparisons.

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

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

Go to exercise #4 to explore this inversion yourself.

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

Do exercises #5 and #6.

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

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

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

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

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

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

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

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

Do exercise #7.

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

Do exercise #8.

2.6 Grouping statements via identation (exercise #9)

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

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

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

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

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

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

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

Do exercise #9.

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

Do exercise #10.

2.7 Checking the answer (Exercise 11)

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

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

Put your code into exercise #11.

2.8 Picking number randomly (Exercise 12)

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

```
from random import randint
```

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

Put your code into exercise #12.

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

2.9 One-armed bandit (Exercise 13)

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

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

Put your code into exercise #13.

Chapter 3

Guess the Number, the Sequel

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

3.1 While loop (Exercises 1-2)

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

The basic structure of a *while* loop is

```
# statements before the loop

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

# statements after the loop
```

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

Do exercise #1.

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

```
from random import randint

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

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

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

Put your code into exercise #2.

3.2 Counting attempts (Exercise #3)

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

Put your code into exercise #3.

3.3 Breaking (and exiting, Exercise #4)

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

```
# this code runs before the loop

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

    if <someothercondition>:
```

```
        break

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

# this code runs after the loop
```

Do exercise #4 to build the intuition.

3.4 Limiting number of attempts via break (Exercise 5)

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

Put your code into exercise #5.

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

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

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

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

Put your code into exercise #6.

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

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

Put your code into exercise #7.

3.7 Show remaining attempts (Exercise 8)

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

Put your code into exercise #8.

3.8 Repeating the game (Exercise 9)

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

```
from random import randint

# define MAX_ATTEMPTS

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

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

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

Pay extra attention to indentation to group the code properly!

Put your code into exercise #9.

3.9 Best score (Exercise 10)

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

Put your code into exercise #10.

3.10 Counting game rounds (Exercise 11)

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

Put your code into exercise #11.

3.11 Wrap up

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

Chapter 4

Hunt the Wumpus, part 1

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

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

4.1 Lists

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

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

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

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

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

Do exercise #2 to build the intuition.

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

Do exercise #3.

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

Do exercise #4.

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

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

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

Do exercise #5.

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

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

Do exercise #6.

4.2 Caves

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

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

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

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

To allow the player to wander, we need to know where 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.

```
...
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.

```
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.

```
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)

```
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

```
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).

```
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).

```
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!).

```
# 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` whe
# 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 unoccupied 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()`. Remember that you can either import the *entire* library and then call its function by prefixing them with the library name or you can import only a specific function via `from ... import ...`.

```
#1  import entire library
import random

random.random()
random.randint(5)

# import only functions you need
from random import randint, random

random()
randint()
```

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 of the loop
            if hit player -> oops game over message, gameover=True, and break out of the loop
        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 (remaining_arrows v
        ask about cave that player want to shoot at (input_cave function), store answer in `arrow
        ask how the far the arrow should fly (input_distance function), store answer in `shooting

        fly arrow through caves in for loop (shooting_distance caves):
            if hit wumpus -> congrats game over message, gameover=True, and break out of the loop
            if hit player -> oops game over message, gameover=True, and break out of the loop
            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 bottom-left corner is (-1, -1) whereas the top-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.

Chapter 9

Memory game, part 1

Today, we will continue learning how to use PsychoPy. To this end, you will write a good old *Memory* game. Sixteen cards are lying "face down", you can turn any two of them and, if they are identical, they are taken off the table. If they are different, the cards turn "face down" again.

Before we start, create a new folder *Memory01* for exercise files and create a subfolder *Images* in it. Then, download images of chicken[1] that we will use for the game and unzip them into *Images* subfolder.

As per usual, we will start with bare basic and will add the complexity along the way. For the main script(s), please use VS Code (or IDE of your choice) but I would still encourage you to use Jupyter to figure out small code snippents.

9.1 Minimal code

The minimal code that we start with is

```
from psychopy import visual, event

win = visual.Window(size=(???, ???), units="norm")

# wait for a key press

win.close()
```

Your first task is to figure the optimal image size. Each chicken image is 240×400 pixels and, for the game, we need place for *exactly* 4×2 images, i.e. our window

[1] The images are from OpenClipart and are public domain.

must be 4 cards wide and 2 cards high. Compute the window size and put into the code. Add event.waitKeys(), to check the window before it closes.

Put your code into *exercise01.py*.

9.2 Drawing an image

Last time we displayed text and rectangles. Today, we will use images (see instructions above on downloading them). Using an image stimulus in PsychoPy is very straightforward. First, you need to create an new object by calling `visual.ImageStim(...)`. You can find the complete list of parameters in the documentation (see the link above) but for our initial intents and purposes, we only need to pass three of them:

- our window variable: `win`
- image file name: `image = "Images/r01.png"`
- size: `size=(???, ???)`. That is one for you to compute. Given that we use `"norm"` units (windows is 2 units wide and 2 units high) and we want to have a 4×2 images, what is the size of each image in `"norm"` units?

Create the simple code that should follow the template:

```
from psychopy import visual, event

win = visual.Window(size=(???, ???), units="norm")

# create the image stimulus, call variable "chicken"

# draw chicken image just like you drew text or rectangle in previous seminar
# flip window

# wait for any key press

win.close()
```

Put your code into *exercise02.py*.

9.3 Python function arguments/parameters

When we created the image stimulus, we used two tricks: passing value by parameter name and using default values.

9.3.1 Position and name (keyword)

In Python, you can pass values to parameters by position or using their name. For example, if you have a simple function

```
def subtract(x, y):
    return x - y
```

you can call it by passing two values to the function `subtract(2, 3)`, which will return `-1`. However, you can also use parameter names to make it more explicit, *e.g.* `subtract(x=2, y=3)`, which will also return `-1`. If you use parameters by name, you do not need to list in the original order, *e.g.* `subtract(y=3, x=2)` will, again, return `-1`.

Moreover, you can mix position and keyword (named) parameters in the same call, *e.g.* `subtract(2, y=3)`. However, the position parameters must always come *before* named parameters `<function>(<value1>, <value2>, <param>=<value>, ...)`. Thus, in case of our simple function above, `subtract(x=2, 3)` won't work.

9.3.1.0.1 Default values When you write a function, you can also specify *default* values for its arguments. This way, when the function is called, you **must** specify the parameters without default values and you **can but do not have to** also pass values to the arguments with the default values. PsychoPy relies heavily on defaults, allowing you to specify a bare minimum. For example, when we created the `ImageStim`, we only specified three parameters: `win`, `image`, and `size`. We **had to** specify the first two, as there are no meaningful defaults for them (PsychoPy needs to know which image and in which window it must use). But we could have left `size` argument out, in which case PsychoPy would use the actual image size (240×400 pixels, in our case).

When writing a function, you simple add `=<default_value>` to the argument, *e.g.*:

```
def subtract(x, y=1):
    return x - y
```

Now, you call `subtract(5)` to get `4` because function will use the default valule for `y`, which is 1. But you can always specify value for `y` as in previous examples `subtract(5, 3)` to get `2`.

Don't forget to document the default values and, preferably, the reason for having these specific values as defaults.

9.3.2 Using *os* library

We specified image file name as `"Images/r01.png"`. This did the job but, unfortunately, major operating systems disagree with Windows on which slash should be used. To make your code more robust, you need to construct a proper filename string using *os* library. It contains various utilities for working with your operating system and, in particular, with files and directories. The function we will need for this task is `join` in path submodule. When you `import os` library, you can then call this function as `os.path.join()`. It takes path components and joins them to match the OS format. E.g., `os.path.join("Python seminar", "Memory game", "memory01.py")` on Windows will return `'Python seminar\\Memory game\\memory01.py'`.

As we will need to load multiple files from the same folder later on, modify the code to make it more universal. First, create a constant `IMAGE_FOLDER = "Images"`, which specifies folder with our images, and a `card_filename = "r01.png"`. Then, modify your `ImageStim(...)` function call by constructing the full filename using `os.path.join` from the images folder and card filename variables.

Put your code into *exercise03.py*.

9.4 Ordnung muss sein!

When you import libraries, all import statements should be at the top of your file and you should avoid putting them there in random order. The recommended order is 1) system libraries, like `os` or `random`; 2) third-party libraries, like `psychopy`; 3) your project modules (we will use them shortly). And, within each section you should put the libraries *alphabetically*, so

```
import os
import random
```

This may not look particularly useful for our simple code but as your projects will grow, you will need to include more and more libraries. Keeping them in that order makes it easy to understand which libraries you use and which are non-standard. Alphabetic order means that you can quickly check whether a library is included, as you can quickly find the location where its import statement should appear.

9.5 Placing an image

By default, our image is placed at the center of the screen, which is a surprisingly useful default for a typical psychophysical experiment that shows stimuli at

fixation (center of the screen). However, we will need to draw eight images, each at its designated location. To make our life simpler in the long run, let us create a function that takes image index (from 0 to 7) and returns a tuple with its location on the screen. Here the sketch of how index correspond to the location:

```
[0 ][1 ][2 ][3 ]
[4 ][5 ][6 ][7 ]
```

Name the function `position_from_index`, it should take one argument (`index`) and return a tuple with coordinates `(<x>, <y>)` in `"units"` coordinates (because this is our chosen coordinate units system). Then, we can then use this tuple to pass the value to `pos` argument of the `ImageStim()`. For example, when called as `position_from_index(0)` it should return `(-0.75, 0.5)`. When called as `position_from_index(6)` it should return `(0.25, 0.5)`, etc.

Think how you compute position from the index given the size of image. My implementation makes use of the floor division operator `//` and modulos, divison remainder `%` operators. The former returns only the integer part of the division, so that `4//3` is `1` (because 4/3 is 1.33333) and `1//4` is `0` (because 1/4 is 0.25). The latter returns the remaining integers, so that `4 % 3` is `1` and `1 % 4` is `0`.

First, write and debug this function in Jupyter. Initially, you do not even need to have a function. You can just set `index = 0` (or some other value) and write the code below. Once it work, you can add the `def position_from_index(index):` part and **document the function**. Once the function works, put it into a new file *utilities.py*, we will use it to implement out custom function without cluttering the main file.

Put your code into *utilities.py*.

9.5.1 Using your own modules

We need to add the definition of `position_from_index` function to our code. However, putting all function into the main file means that we would have **a lot of code** is a single file. This make it harder to navigate and to read. Thus, we will put our utility functions into a separate module and will import them, just like you import functions from other libraries.

By now the code should be in *utilities.py*. So, you need toimport the function from the module as `from utilities import position_from_index`, this way you can call the function directly as `position_from_index(...)`. Alternatively, you can import entire module as `import utilities` and then call the function as `utilities.position_from_index()`. Both approaches are valid and your preference should depend on the readability of the code in each case.

Now that we have the `position_from_index` function in `utilities.py` file and we imported it, use it to place your image at one of the location. For

this, you just need to add `pos=position_from_index(<index value>)` to the `ImageStim` call. Use different hard-coded values (*e.g.*, `0`) for the index value and see whether image does appear at the correct position.

Put your code into *exercise04.py*.

9.6 Back side of the card

So far, we have an image of the face, which will be a front side. For the game, we also need the back of the card. For this, create a rectangle (Rect stimulus we used the last time) with same width/height as an image and position computed from an index (just like for the image). Pick a combination of a `fillColor` (inside) and `lineColor` (contour) that you like. Modify your code, to draw image (front of the card) and rectangle (back of the card) side-by-side (*e.g.*, if face is at position with index 0, rectangle should be at position 1 or 4). This way you can check that sizes match and that they are position correctly.

Put your code into *exercise05.py*.

9.7 Dictionaries

Below, we will use a dictionary to store all relevant card information and stimuli in a single variable. Dictionaries in Python allow you to store information using *key-value* pairs. This is similar to how you look up a meaning or translation (value) of the word (key) in the real dictionary, hence the name. To create a dictionary, you use *curly* brackets `{<key1> : <value1>}, {<key2> : <value2>, ...}` or create it via `dict(<key1>=<value1>, <key2>=<value2>, ...)`.

```
book = {"Author" : "Walter Moers", "Title": "Die 13½ Leben des Käpt'n Blaubär"}
```

now you can access or modify each field using its key. *E.g.* `print(book["Author"])` or `book["Author"] = "Moers, W."`. You can also add new fields by assigning values to them, *e.g.* `book["Publication year"] = 1999`. In short, you can use a combination of `<dictionary-variable>[<key>]` just like you would use a normal variable.

Please note that dictionaries are mutable - sticker-on-a-object - variables, just like lists. Take another look at mutable values section, if you forgot the implications of this.

9.8 Using dictionary to represent a card

Our card has

1. front side (image of a chicken),
2. back side (rectangle),
3. identity on the card (filename),
4. information about which side is up.

We need #3, so we can later check whether the player opened two identical cards (their filenames match) or two different ones. We need #4 to know how we should draw it (remember, we will have eight cards to manage by the end of the game). Since all of this information belongs to the same card, it would make sense to store it in a single object or a *dictionary*. For didactic purposes of learning how to use dictionaries, we will use the latter.

Create a dictionary variable (name it `card`) with the following fields: * `"front"`: assign your image stimulus to this field (we used to store in a `chicken` variable). * `"back"`: assign your rectangle stimulus to it (make sure that now rectangle is at the same position, as the front of the card). * `"filename"`: filename use used for it. * `"index"`: position index of the card (we will use it later for the interaction), pick the one you like. * `"side"`: assign it initially to be `"back"`.

Now, you have both side and all the information you need in the dictionary. Create a code that draws the side of the card as specified in `"side"` field. Note, you **do not need an if-statement for this**! Think how you do it using the power of dictionaries.

Put your code into *exercise06.py*.

9.8.1 Card factory

So far, we have been creating the card dictionary within our main code. However, at a certain point of time, we will need to create eight of them. Thus, a better idea would be to spin this code off as a function which takes a window variable, filename, and position index arguments and returns a dictionary, just like the one we created. You are effectively just wrapping the already working code into a function. Call this function `create_card` and put it into our `utilities.py` file (don't forget to document it!). Import and use it in the main script, replacing the card dictionary calls with a single function call. Note that now you need the `IMAGE_FOLDER` variable in *utilities.py*, rather than in the main file. Also, think about libraries you will now need import in *utilities.py*.

Put your code into *utilities.py* and *exercise07.py*.

9.8.2 Adding presentation/inputs loop

In our game, the player will click on a card to "turn it around". We will implement a mouse interaction shortly but, first, modify the code to have the main presentation loop as we did on previous seminar. E.g., with `while not`

`gameover:` loop and checking whether the player pressed `"escape"` key to exit the game.

Put your code into *exercise08.py*.

9.8.3 Detecting a mouse click

Before you can use a mouse in PsychoPy, you must create it via `mouse = event.Mouse(visible=True, win=win)` call, where `win` is the PsychoPy window you already created. This code should appear right after your created the window itself.

Now, you can check whether the left button was pressed using `mouse.getPressed()` method. It returns a three-item list with `True`/`False` values indicating whether each of the three buttons are pressed. Use it the main loop, so that if the player pressed *left* button (its index in the returned list is `0`), you change `card["side"]` to `"front"`.

If you run the code and click *anywhere*, this should flip the card.

Put your code into *exercise09.py*.

9.9 Position to index

Currently, the card is flipped if you click *anywhere*. But it should flip only when the player clicked on that specific card. For this we need to implement a function `index_from_position` that is inverse of `position_from_index`. It should take an argument `pos`, which is a tuple of `(<x>, <y>)` values (that would be the mouse position within the window), and return an **integer** card index. You have float values (with decimal points) in the `pos` argument (because it ranges from -1 to 1) and by default the values you compute from them will also be float. However, the index is integer, so you will need to wrap it in `int(<value>)` conversion call, before returning it.

I would recommend debugging the code in Jupyter first. Just set `pos = (-0.9, 0.9)` (index is, then, 0) or some other values within -1..1 range) and make sure it computes a valid index. Once it works, turn it into a function, test it in Jupyter, document it(!), and copy-paste to `utilities.py` file.

Put your code into *utilities.py*.

9.10 Flip on click

Now that you have function that returns an index from position (don't forget to import it), you can check whether the player did click on the card itself. For

this, you need to extend the card-flipping code inside the *if left-mouse button was pressed* code.

You can get the position of the mouse within the window by calling `mouse.getPos()`. This will return a tuple of `(x, y)` values, which you can pass to your `index_from_position()` function. This, in turn will return the index of the card the player click on. If it *matches* the index of your only card (stored in `"index"` field of the `card` dictionary), then and only then you flip the card.

Put your code into *exercise10.py*.

9.11 Flip-flop

As a final exercise for today, let us make the card flip-flopping back and forth. We won't really use this code for the full version of the game but it will let you learn conditional assignment and `clock.wait()` function, you will need to use later.

In order for the card to flip-flop, you need to modify your `card["side"] = "front"` statement, so that it is not `"front"` but *the other* side of the card, which becomes active. There are several ways to implement this but use an if statement that checks the *current* state of the card and assigns *the other* one. So, if `card["side"]` is current `"front"` it should become `"back"` and vice versa. Write and debug this four lines of code in Jupyter. Create card variable in the cell itself, *e.g.* `card = {"side" : "front"}`.

As you are only assigning one of the two values to the same variable inside if-statement, you can use a nifty conditional-assignment to simplify the code. The four-line code below

```
if y == 1:
    x = 2
else:
    x = 3
```

is equivalent to

```
x = 2 if y == 1 else 3
```

Use this conditional assignment to implement flip-flopping of the card. First, test it in a Jupyter cell below and then copy-paste it to your code, replacing `card["side"] = "front"` statement. It will work kinda weirdly, do not worry about this and read on!

By now the flip-flopping should work and, like, *real fast*! This is because `mouse.getPressed()` tells us whether the mouse button is pressed right now.

Even if you are very fast in clicking it, you still hold it for a few frames (dozens of milliseconds) before releasing it. Hence, your card-flipping code is also invoked multiple times. There are several way to solve this problem. One would be to create a `is_pressed` variable to keep track of whether the button was already pressed or released and act accordingly. Here, we will implement a simpler quick (and, admittedly, very dirty) solution. We will simply pause the code for 0.1 seconds giving the player enough time to release the button. This won't help the player who stubbornly keeps holding the button down (then the card will flip-flop every 0.1 seconds) but will take care of simple clicks.

For this, you need to use wait() function in `clock` module of PsychoPy. Import the module (alongside `visual` and `event`) and call this function right after you flipped the card. Use `0.1` seconds waiting time but you can experiment using shorter or longer ones.

Put your code into *exercise11.py*.

9.12 To be continued...

Well done! By now, we have the code that creates a card with a given image and at a given position, we can draw it based on which side should be shown, and we can detect when the player clicked on that card. Next time, we will use these abilities to add more cards and turn it into a real game.

Chapter 10

Memory game, part 2

By the time we finished our previous seminar, you had the code that created a single "card" with a given image at a given location and you were able to interact with it (flip-flopping it). Now, we need to extend the code so that we have eight cards that we can open only two at a time. If cards match (have same `"filename"` field), we remove them. If cards do not match, we simply turn them over.

Before we start, create a new folder *Memory02* for exercise files. Copy *Images* subfolder, as well as the `utilities.py` and *exercise11.py* (rename it to *exercise01.py*, as we will use the latter as the staring point). Also, download the Jupyter notebook, which you will use to experiment with functions and do exercises. You will need to upload it along with other files.

10.1 Getting list of image files.

For a single card, we simply hard-coded the name of the image file, as well as its location. However, for a real game (or an experiment) we would like to be more flexible and automatically determine which files we have in the *Images* folder. For this, you need to use os.listdir(path=".") function that, you've guess it, returns a list with filenames of *all* the files in a folder specified by path. By default, it is a current path (`path="."`). However, you can use either a relative path - `os.listdir("Images")`, assuming that *Images* is a subfolder in your current directory - or an absolute path `os.listdir("E:/Teaching/Python/MemoryGame/Images")` (in my case). In Jupyter, write a single line code to get the list of files in the *Images* folder. Use *relative* path, if it is in a subfolder relative to your Jupyter notebook, or use an *absolute* path, if it is in a different folder. Do not forget to `import os` before you run the code, of course!

Do exercise #1 in Jupyter notebook.

You should have gotten a list of 8 files that are coded as *[r/l][index].png*, where r or l denote the side the chicken is looking at. However, for our game we need only four images ($4 \times 2 = 8$ cards). Therefore, we need to select a subset of them. For example, a random four or chicken looking to the left or to the right only. Here, let us work with chicken looking to the left, meaning that we need only to pick files that start with "l". To make this list filtering more efficient, we will use list comprehensions.

10.2 List comprehension

List comprehension provide an elegant and easy-to-read way to create, modify and/or filter elements of the list creating a new list. The general structure is

```
new_list = [<transform-the-item> for item in old_list if <condition-given-the-item>]
```

Let us look at examples to understand how it works. Imagine that you have a list `numbers = [1, 2, 3]` and you need increment each number by 1[1]. You can do it by creating a new list and adding 1 to each item in the part:

```
numbers = [1, 2, 3]
numbers_plus_1 = [item + 1 for item in numbers]
```

Note that this is equivalent to

```
numbers = [1, 2, 3]
numbers_plus_1 = []
for item in numbers:
    numbers_plus_1.append(item + 1)
```

Or, imagine that you need to convert each item to a string. You can do it simply as

```
numbers = [1, 2, 3]
numbers_as_strings = [str(item) for item in numbers]
```

What would be an equivalent form using a normal for loop? Write both versions of code in Jupiter cells and check that the results are the same.

Do exercise #2 in Jupyter notebook.

Now, implement the code below using list comprehension. Check that results match.

[1] A very arbitrary example!

```
strings = ['1', '2', '3']
numbers = []
for astring in strings:
    numbers.append(int(astring) + 10)
```

Do exercise #3 in Jupyter notebook.

As noted above, you can also use conditional statement to filter which items are passed to the new list. In our numbers example, we can retain numbers that are greater than 1

```
numbers = [1, 2, 3]
numbers_greater_than_1 = [item for item in numbers if item > 1]
```

Sometimes, the same statement is written in three lines, instead of one, to make reading easier:

```
numbers = [1, 2, 3]
numbers_greater_than_1 = [item
                          for item in numbers
                          if item > 1]
```

You can of course combine the transformation and filtering in a single statement. Create code that filters out all items below 2 and adds 4 to them.

Do exercise #4 in Jupyter notebook.

10.3 Getting list of relevant files

Now that you know about list comprehensions, you can easily create a list of files of chicken looking left, *i.e.* with filenames that start with "l". Use str.startswith() for filtering, store the list in `filenames` variable. Put you code into a Jupyter cell.

Do exercise #5 in Jupyter notebook.

This gives us a nice list of eight files but we need each name twice. There are several ways of making but we will use list operations for this.

10.3.1 List operations

Python lists implement two operations:

- Adding two lists together: `<list1> + <list2>`.

```
a = [1, 2, 3]
b = [4, 5, 6]
a + b
```

gives a *new* list with items `[1, 2, 3, 4, 5, 4, 5, 6]`. Note that this is not equivalent to extend method `a.extend(b)`! The `+` creates a *new* list, `.extend()` extends the original list `a`.

- List multiplication/replication: `<list> * <integer-value>` creates a *new* list by replicating the original one `<integer-value>` times. For example:

```
a = [1, 2, 3]
b = 4
a * b
```

will give you `[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]`.

Use either operations or `.extend()` method to create the new list where each filename is repeated twice. Hint, you can apply list multiplication directly to the filenames list you created via list comprehension.

Do exercise #6 in Jupyter notebook.

10.4 Lots of cards, using list enumeration

Now that we have a list of filenames, we can create a list of cards out of it. This list will replace your single `card` variable. Name the new list `cards` and create it either using an empty list + for loop or using list comprehensions. Inside the loop you creating each card using its filename and index (see below) and append it to the `cards` list (or it is added automatically, if you use comprehensions).

You can get the index of each filename (and, therefore, it position on the screen) via enumerate(), which is a generator that returns a tuple of `(index, item)`. For example, test the example below that prints one letter at a time with its index:

```
letters = ['a', 'b', 'c']
for index, letter in enumerate(letters):
    print('%d: %s'%(index, letter))
```

Once you created the list of cards, you should draw *all* of them inside the main loop. You already have a line that draws a single `single_card`. You should wrap this call with a for loop that loops over all `cards` and uses `single_card` as an iterator variable. Your code should look roughly as this

```
# import libraries
# open window and create the mouse
# get filenames for chicken looking left
# create all cards

gameover = False
while not gameover:
    # draw all cards

    # check keys

    # mouse processing - COMMENT IT OUT FOR A MOMENT
```

Put your code into *exercise01.py*.

10.5 Mouse interaction for every card

Our mouse interaction was based on the index of our only card and checking that it matches the mouse click. However, now we have eight cards that cover the *entire* window. Thus, it does not matter where exactly the player clicked, there is a card which needs to be turned over. Modify your mouse click processing code so that if mouse left button is pressed, it computes an index of that location (it would be from 0 to 7), flips the card with that index in the list, waits for 0.1 seconds (our hack from last time, we won't need it soon).

Put your code into *exercise02.py*.

10.6 Limiting flipping to just two cards

In the actual game, we are allow to only turn around *two* cards at a time. We will either need an extra variable to know how many cards are face up already or do an on-the-fly scan through the list. Let us implement the extra variable solution. Create a new variable `faceup_n = 0` before the loop. Inside the mouse-click processing code, you need to implement the following logic:

```
if mouse.getPressed()[0]:
    # compute index of the mouse click
    # if card with this index is not face up already:
        # turn the card face up
        # increment the faceup_n variable
        # if faceup_n is equal to 2:
            # --- draw all cards ---
            # insert a pause of 0.5 seconds
```

```
            # turn all cards back
            # reset faceup_n to zero (as all cards are now with their backs up)
```

Note that we need to explicitly draw the cards again inside the condition (and don't forget to flip the window)! Otherwise, we turn the card back before we ever get to the drawing stage.

Put your code into *exercise03.py*.

10.7 Remembering which cards were turned

Our initial implementation was to count the number of cards that player turned. This, however, means that we don't know which cards these were. Let us now store face-up cards in a new list called `faceup` that would replace the `faceup_n` variable. Initialize `faceup` with an empty list (just like you initialized `faceup_n` with 0). And, once the card is flipped, add it to this list. Once `faceup` list is 2 items long (use len() function to get the length of the list), show the cards, wait for 0.5 seconds, turn cards in `faceup` list back, and clear the list.

```
if mouse.getPressed()[0]:
    # compute index of the mouse click
    # if card with this index is not face up already:
        # turn the card face up
        # add card to faceup list
        # if faceup list length is equal to 2:
            # --- draw all cards ---
            # insert a pause of 0.5 seconds
            # turn cards in faceup list back
            # clear faceup list
```

Put your code into *exercise04.py*.

10.8 "Visible" card flag

In our next step, we will remove the matching cards. However, in reality, we won't actually remove them, we just won't draw them and won't allow user to interact with them once they are "removed". To know whether we need to draw a particular card, we will add a new field `visible`. There are three (well, four) pieces of code which needs to be modified.

1. Add `visible` field to the dictionary in the `create_card()` function when making the card and set it to `True` by default.

2. When drawing cards, draw it only if `visible` field is `True`. By now you draw card in *two* places, so both need to be appended.
3. In the code that processes the mouse click, when checking if the card hasn't been turned over yet, add "and it is visible" to the condition (in Python syntax, of course). So that it can be turned face up only if it is current face down and visible.

You should see no difference in the game, because as long as all cards are visible (and, at the moment, they always are) it works just like before.

Put your code into *utilities.py* and *exercise05.py*.

10.9 "Removing" matching cards

By now we now which two cards were turned over (these are in `faceup` list) and if they match, which means that their filenames match, we can "remove" them by setting their `visible` field to `False`.

```
# if faceup list length is equal to 2:
    # --- draw all cards ---
    # insert a pause of 0.5 seconds

    # if cards match:
        # set cards visibility to False
    # else:
        # turn them back

    # clear faceup list
```

Test this and remember that we didn't shuffle the faces so they repeat in an orderly fashion, making it easy to open the pairs.

Put your code into *exercise06.py*.

10.10 Game over, if you run out of cards

Currently, our game can be exited by pressing *Escape*. Let's add a more positive game-over outcome when you win because you matched all the cards! For this, we need to count how many cards are left on the table and modify the loop, so that it iterates only as long as the number of cards left is more than zero. For this,

1. Create a new variable `cards_left` before the loop start and set it to the total number of cards (length of the cards list).

2. Modify the while loop, adding the second condition that `cards_left` must be more than zero.
3. Decrease `cards_left` by 2, every time two cards are matched and "removed".

Put your code into *exercise07.py*.

10.11 Game over message

Currently, our game abruptly closes once all cards are removed. It would be much friendlier to add a simple "Game over" message and show it for couple of seconds. After the main loop but before closing the window, create and draw the [TextStim]((https://psychopy.org/api/visual/textstim.html) and use either `clock.wait(2)` or `event.waitKeys()` (your choice). However, make sure that this message is only shown if there were no cards left. Otherwise it was the escape button press and so no "game over" message for people who are too lazy to finish the game!

Put your code into *exercise08.py*.

10.12 Counting attempts

It would be even more fun to check how many attempts it required for the player to match all cards. Create a new variable `attempts` and initialize it to zero before the main loop. Increase it by 1 every time the player turned two cards over (irrespective of whether they matched). Add the number of attempts to the "Game over" message.

Put your code into *exercise09.py*.

10.13 Record time

Let us record the time it took the player to complete the task. For this we will use PsychoPy's Clock class, which is very straightforward. Simply create the `timer = clock.Clock()` before the main loop and then call its getTime() method to get the elapsed time in seconds. Add this information to the "game over" message.

Put your code into *exercise10.py*.

10.14 Randomness

The only boring thing about our game is that we know exactly where each chicken is. Shuffle the filenames list before you create the cards to turn it into a proper memory game.

Put your code into *exercise11.py*.

Yay! Awesome game!

Chapter 11

Memory game, part 3

During our previous seminar, we competed the core **Memory Game**. Today we will put more bells-and-whistles on it.

Before we start, create a new folder *Memory03* for exercise files. Copy *Images* subfolder, as well as the `utilities.py` and *exercise11.py* (rename it to *exercise01.py*, as we will use the latter as the staring point) from the previous seminar. Also, download sounds[1] file and unzip it into a *Sounds* subfolder in the seminar directory.

11.1 Sound effects

Let us add "clicking" sounds whenever the player turns the card. For this, we will use sound module of PsychoPy library. If your sound does not play, ask me and we will try to set your sound libraries up.

First, you need to import the `Sound` class as suggested in the manual:

```
from psychopy.sound import Sound
```

Next, create a new sound and assign it to a variable `click_sound`, just like you created the visual stimuli. Use a note name (*e.g.* `"C"` or `"A"`) and a short duration (*e.g.*, `0.1` or `0.2` seconds). Do this at the beginning, right after we have created the mouse.

Finally, you need to `.play()` this sound just like you `.draw()` the visual stimuli. However, in PsychoPy by default the sound will not rewind back to the beginning after it finished playing (at least not for the generated sound and the

[1] The files are public domain from freesound.org.

setup I have). So once you play it once, it is "at the end" and the next time you try to play it, there will be no sound (because it has finished playing already). To "rewind" the sound to the beginning, you first `.stop()` it and then, immediately, `.play()` it (a bit counterintuitive but works). Do it when the player turns the card over (not just clicks somewhere).

Put your code into *exercise01.py*.

11.2 Sound from file

Now, let us use more complex prerecorded sounds. Put the files from the **sounds.zip** folder into a *Sounds* subfolder of the Memory game. Mixing generated sounds (like we used in the example above) with prerecorded does not always work and it is the limitation of the libraries that PsychoPy relies upon. Thus, let us replace the way the create the `click_sound` by using *click.wav* instead of the note value. Note that you must specify the path to that file (it is in the `"Sounds"` folder), so use use `os.join.path()`.

Interestingly, at least for my setup, the sound it reset automatically, so you do not need to `.stop()` it before you `.play()` it. Thus, you can drop that extra `.stop()` call.

Put your code into *exercise02.py*.

11.3 Feedback sounds

Now let us add two more sounds: `correct_sound` variable that uses `"correct.wav"` and `error_sound` that uses `"error.wav"`. They should be played when two cards are opened, right before the delay. Play `correct_sound` if the cards matched, `error_sound` otherwise.

```
if two cards are opened:
    draw cards

    if cards match:
        play correct_sound
    else:
        play error_sound

    clock.wait(0.5)
```

Put your code into *exercise03.py*.

11.4 Keeping sounds organized: dictionary comprehension

We now have three variables for three different sounds. We could be tidier than that by putting them into a dictionary `sounds` and using them as `sounds['correct'].play()`. This is a more general, tidier (just one variable rather than lots), and more future-proof approach. You can create this dictionary in a direct way `sounds = {"click" : Sound(...), ...}` but will learn and use dictionary comprehension instead.

Dictionary comprehensions are very similar to list comprehensions. You also loop over a list but you use `{}` instead of `[]` and you use an item to generated both a key and the value for a dictionary entry. For example:

```
index_letters = {i : chr(65+i) for i in range(4)}
```

Here, you use the item as a key `i` and generate a string value for it. Note that although keys **must be unique**, the values are not, so you can use the same value for all entries. Imagine that you are keeping the score in the game and all players start with a zero:

```
players = ['Anna', 'Bob', 'Chris']
score = {player : 0 for player in players}
```

Change the code to generate `sounds` dictionary via comprehension and use it instead of variables. In our case, the three keys should be `"click"`, `"correct"`, and `"error"`. You can generate the filename from these keys, just do not forget to add the `".wav"` extension.

Put your code into *exercise04.py*.

11.5 Logging data

Although we recorded basic stats like the number of attempts or the total amount of time a player required, for a real experiment you would want to record as much information as possible. In our case, we would want to record every card flip. In the real experiment, that would inform us which chicken that are easier or harder to memorize.

For this, we will use ExperimentHandler, which is a part of experimental scheduling and data logging tools of PsychoPy. Basically, we need to have three additional pieces of code: 1) one that creates the handler, 2) one that logs the information about the click (card, time, and attempt number), 3) one that save the log to a file.

1. Create an `ExperimentHandler()` object and assign it to the `exp` variable. Do it near the place where you created the window. There are multiple options you can specify but for out intents and purposes the defaults will do.
2. To log information about each click, you need to first .addData(key, value) for each variable (key) and value you want to record. In our case you need to call it three times for `"card"` (value is the `"filename"` of the card the player clicked on), `"attempt"` (value is the index of the current attempt), and `"time"` (value is the time since the start, you already have a `timer` variable that you can use for this). After all three `.addData()` calls, you need to tell the handler to advance to the next row of you log table using .nextEntry() method. Put this code right after your "flipped" the card.
3. To save information to a file, you need to call .saveAsWideText(…) method of the `exp`. You need to supply the `filename` parameter and, possibly, further options such as a delimiter (`delim`). My suggestion would be to use a filename such as `"log.csv"` and `","` for a delimiter symbol (this would make a standard comma-separated-values file that any program can read). Put this code at the very end after the game over message.

Put your code into *exercise05.py*.

11.6 More rounds

We have been using left-facing chicken (the image files that start with `"l"`) but we also have right-facing chicken (the image files that start with `"r"`). Let us use both sets, so that games run for two rounds, first left- then right-facing chicken.

For this, we will need to create a variable with the list of conditions. Let us, inventively, call it `conditions` and it will be a simple list of two items: `"'l'"` and `"r"`. Create this variable in the beginning, **before** you create cards.

Next, create an outer loop over the conditions. You code should look roughly like that

```
imports
creating window, mouse, sounds, experiment handler, etc.
create conditions list
for condition in conditions:
    create cards using condition letter to filter files
    prepare everything for the round (gameover variable, timer, attempts counter, etc.)
    while not gameover:
        visuals
        mouse handling
        keyboard handling
```

```
    game over message (change it to block results message)

save logs
close window
```

When you create cards use the `condition` (assuming that you used `for condition in conditions:`) instead of the hardcoded `"1"` to filter the files. You also to need to log the condition. Add a `exp.addData()` call for it along the other ones.

Finally, modify your "game-over" message to be "end-of-the-block" message. You still should show the stats, change only with wording.

Put your code into *exercise06.py*.

11.7 Random order

Randomize order of conditions before the main loop.

Put your code into *exercise07.py*.

11.8 ABBA order

When running an actual experiment, you may want to account for the learning effect. If conditions are presented in separate blocks (as in our case), the ones that occur later will benefit from learning (although they will also suffer because of fatigue). One way to even things out, is present conditions twice in a direct and, then, a reverse order (ABBA). This way, a condition which was used first (no learning) is also the last one (maximal learning).

Once you randomized the order of the first two blocks, you need to add the same list but in a reverse order. One possibility is a .reverse() method. However, it reverses the list itself in-place. Instead you should use slicing to create a *new* reversed list before adding it to the original one. Read again on lists, if you forgot how it works. The idea is simply to `conditions = conditions + conditions-that-are-sliced-to-be-in-reverse-order`. You can test this code in a Jupyter cell, before putting into the main code.

Put your code into *exercise08.py*.

11.9 We want more!

Now you have a real working experiment with randomized condition order, logging, feedback, etc. Think about how you can improve it further. Instructions? More/less feedback?

Chapter 12

The skill of programming

12.1 Writing the code

Programming is not about writing code that works. That, obviously, must be true but it is only the minimal requirement. Programming is about writing a clear easy-to-read code that others and, perhaps even more importantly, you-two-weeks later can understand. Below are several suggestions on how you can improve your code.

- **Use a linter.** In VS Code, press *Ctrl + Shift + P* and start typing `Python: Select linter`. You will be offered a number of choices, pick *pylint* (that's the one I use) or *flake8* (it is more strict). VS Code will offer you to install the actual linter package, if it is missing. Then, the linter will analyze your code after every save and highlight issues it finds: spaces where should be none, no spaces where should be some, wrong names, trailing spaces, overly long lines, etc. Try to address all the problems as it will make you code look consisten, "standard", and, therefore, readable. However, use your better judgment because sometimes longer-than-linter-likes line is more readable than two shorter ones. Similarly, a "bad" variable name by linter standards can be a meaningful name for a psychologist. Remember, your code is for people, not for the linter.
- **Document your code.** Every time you create a new function: document it. New class: document it. New constant: unless it is super clear from the name alone, document it. First, VS Code is smart enough to parse Numpy docstring on the fly, so it will show this help to you whenever you use your own functions. Second, it forces you to think and formulate (in human language!) what the function is doing, what type the arguments are, what is the range of valid values, what are the defaults, what should it return, etc. More often than not, you will realize that you have overlook some important detail that is not apparent from the code itself.

- **Add some air.** Separate chunks of code with some empty lines. Think paragraphs in the normal text, you wouldn't your book to be a single paragraph nightmare? Put a comment before each chunk that explains what it does but not *how* it does it. E.g., in our typical PsychoPy-based game there is a point when we draw all stimuli and flip the window buffers. That is a nice self-contained chunk that can be described as `# drawing all stimuli`. The code provides details on what exactly is drawn, what is the drawing order, etc. But that single comment will help you understand what this chunk is about and whether it is relevant for you at the moment. Same goes for `# processing key presses` or `# checking gameover conditions`, etc. But be careful and make sure that the comment describes the code correctly. E.g., if the comment says `# drawing all stimuli` where should be no stimuli-drawing code anywhere else!
- **Write your code one teeny-tiny step a time.** Your motto should be "Slow but steady". This is the way I guide you through the assignments. Start with a something simple like a static rectangle or image. Make sure it works. Add a minor functionality (change in color, position, another rectangle, storing it as an attribute, etc.). Make sure it works. Never go to the next step unless you fully understand what your current code is doing and you are 100% certain that it behaves as it should. This tortoise-speed approach may feel silly and overly slow but it is still faster than writing a large chunk of code and then trying to make sure that it all works. It is much easier to solve simple problems one at a time than a lot of them simultaneously.

Unfortunately, these tricks won't work it you do not use them! So you should *always* use them and they should become your *good habits*, like using a seat belt. It does nothing on most (hopefully, all) days but you wear it because it might suddenly become extremely useful and you can never be sure when this will happen. Same with coding. Quite often you will be tempted to write "quick-n-dirty" code because this is just a "simple test", temporary solution, a prototype, pilot experiment, etc. But, as they say in Russia "There is nothing more permanent than a temporary solution", and you may find that your toy code grew into a full blown experiment and it is a mess. Or you want to come back to that pilot experiment but realize that it is easier to start from scratch than to understand how that monster works[1]. Thus, resist the temptation! Form the good habits and you future-you will be very grateful!

12.2 Reading the code

My experience with programming in general and on this seminar in particular is that most problems you get stuck with are simple (to be point of being dumb)

[1] Happened to me more often than I dare to admit.

and obvious in retrospect[2]. Unfortunately, this knowledge is of little comfort while you are trying to figure out why the bloody thing does not work! Here are several suggestions that could help you to resolve them faster.

- **Think like a computer.** Read the code line-by-line and "execute" it the way the compute would. Use pen-and-paper to keep the track of variables. Trace which chunks of code can be reached when. Slow yourself down and make sure you understand each line and are able to keep track of the variables. Once you do that it will be easy to spot a mistake.
- **Pretend that you've never seen this code in your life.** Assume that you have no idea what it is doing. Quite often you *literally* do not see a mistake because your brain fills-in details and bends the reality to match your expectations. You *know* what this chunk of code should be doing, so instead of reading it you skim through it and, unless it looks obviously terribly wrong, assume that it does what it should. This is not just programming-specific thing but a general feature of our perception. If you ever proof-read your writing for typos, you know how mind-boggling difficult it is to spot them[3]! Turning your expectations off is hard but is immensely helpful.
- **Use the debugger.** Put breakpoints and execute the code step-by-step. Check values of variables using "Watch" tab. Use debug console to check whether functions return the results that they should. For complex conditions or mathematical formulas, split them into small bits, copy and execute these bits in the debug console and check whether numbers add up. Make sure that a code chunk checks out and then proceed to analyze the next one. Debugging is particularly helpful to identify the code that is not reached or reached at the wrong moment.

12.3 Zen of Python

I also found Zen of Python to be good inspiration on how to approach programming.

[2] Hindsight is always 20/20!

[3] Advice, read each paragraph backwards: Last sentence, penultimate sentence, etc. This breaks the flow of the text and helps you concentrate on words rather than on the meaning and the story.

Chapter 13

Snake game

Welcome back! The purpose of today's seminar to refresh your knowledge of Python acquired previously. We will use dictionaries and lists, as well as conditional statements and loops. Plus, you will need to write functions.[1]

13.1 Assignments

For this and following projects, use Python IDE 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 01* for today) and create a separate file (or files, later on) for each exercise[2] (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 comment on them. For submitting the assignment, just zip the folder and upload the zip-file.

13.2 Snake game: an overview

Today, we will program a good old classic: the snake game! The story is simple: you control a snake trying to eat as many apples as you can. Every time you

[1] Sneaky preview: You will start learning about object-oriented programming during our next seminar and we will turn it into object-based game.

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

consume an apple, snake's length increases. However, if you hit the wall or bite yourself, the game is over (or you lose one of your lives and game is over once you run out of lives).

Here is how the final product will look like.

As before, we will program the game step by step, starting with an empty gray PsychoPy window. Here is the general outline of how we will proceed:

1. Create boilerplate code to initialize PsychoPy.
2. Figure out how to place a square. We need this because our snake is made of square segments and lives on a rectangular grid made of squares.
3. Create a single segment stationary snake[3].
4. Make the snake move assuming a rectangular grid.
5. Implement "died by hitting a wall".
6. Add apples and make the snake grow.
7. Add check for biting itself (and dying).
8. Add bells-and-whistles to make game look awesome.

As you can see, each new step builds on the previous one. Because of that, remember, do not proceed to the next step until the current one works and you fully(!) understand what each line of code does. Any leftover uncertainty will linger, grow and complicate your life disproportionately!

13.3 Initializing PsychoPy

To remind yourself on how you initialize PsychoPy window, see here. Let us plan ahead and decide on windows size and its units. Recall that PsychoPy has five different units for size and position. Given that our snake will be composed of square segments and move on a grid made out of squares, which units should we pick? Read on units and think which units you would pick before continuing

---

My suggestion would be `"norm"` units as they make sizing squares easy, as long as we use a suitable aspect ratio. Thus, let us think about the grid. We can define its width and height in *squares*, e.g., a 30×20 grid should give us enough space to try things out but we can always increase the resolution of the game later. Create a constant `GRID_SIZE` and assign a tuple of `(width, height)` to it. Note that window's aspect ratio will depend on the size of the grid that we pick. If you want to have a square window, use a 30×30 grid.

We also need to define an *absolute* size of a square in pixels (call the constant `SQUARE_SIZE_PIX`), which will determine how large each square will look

[3]Not very exciting, I know. But one has to start somewhere!

on the screen and, therefore, will determine the absolute size of the window: *window height = window height in squares * square size in pixels* (same goes for the width). Note that this parameter determines how the game looks, double the size of the square in pixels and that will double both width and height of the window.

Put it all together. You could should look roughly as follows:

```
# import all libraries and modules you need

# define constants GRID_SIZE and SQUARE_SIZE_PIX

# create PsychoPy window, computing it size in pixels from the two constants

# wait for any key press (just so that window stays on the screen)

# close PsychoPy window
```

Experiment with different grid and square sizes and pick the one that fits your screen.

Put your code into *exercise01.py*.

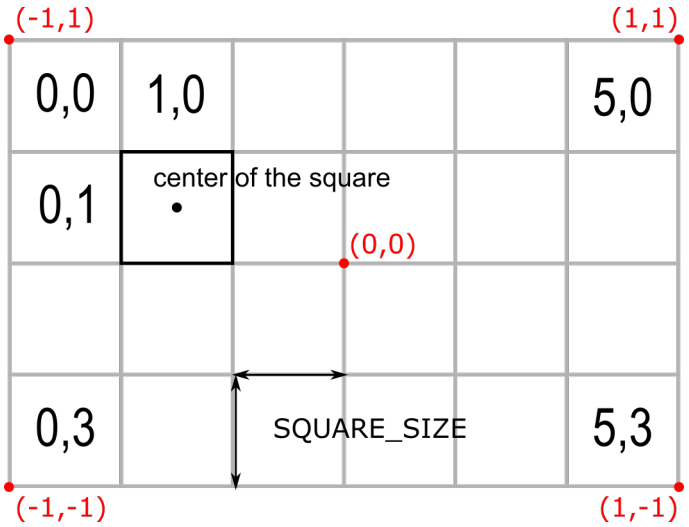
13.4 Adding a square

As I already wrote, our game will be made of squares. Snake is made of squares. Grid it lives on is made of squares. An apple is a square. Which means we need to know the size of that square in the units of the window and we need to know *where* should each square go in the window, based on its coordinates on the grid.

Computing the size of the square in units of window size is easy, so let us start with it first. If you used `"norm"` units for the PsychoPy window, we know that both its width and its height go from -1 (bottom) to 1 (top). We also know that we have to fit in `GRID_SIZE[0]` squares horizontally and `GRID_SIZE[1]` vertically (recall that `GRID_SIZE` defines the size of our grid in number of squares). Create a new constant `SQUARE_SIZE` that is a *tuple* and compute square size in units of `"norm"` based on total width/height of the window in the `"norm"` units (not the pixels!) and number of squares we need to fit in horizontally/vertically.

Next, let us create a function that maps a position on the grid to a position in the window. This way, we can think about position of the snake or apples in terms of the grid but draw them in the window coordinates. Create a new file *utilities.py* and create a function `map_grid_to_win()` that takes a tuple of integers `(x_index, y_index)` with grid position (I called this argument `ipos`) plus a second parameter with a square size (that we just computed) and returns

a tuple of floats `(x_pos, y_pos)` with the coordinates of the *center* of the square in *window* coordinates. Take a look at the drawing below to see the logic of the computation. The red text show location of red points in *window norm* units, whereas the black font shows location of a square in *grid index* units. Note that you need to compute where the *center* of the square should go.



I suggest using Jupyter notebook to create and debug this function and then copying it into *utilities.py*. Remember to document the function following NumPy docstring format.

Put function map_grid_to_win into *utilities.py*.

Now, test this function by creating a square (you remember which constant defines its size, right?) and placing it at different locations of the grid. Check here, if you forgot how to create squares in PsychoPy. Run the code several times, using different grid indexes or adding several squares to check that it work as intended. Your code should look roughly as follows

```
# import all libraries and modules you need

# import map_grid_to_win function from utilities.py file

# define constants GRID_SIZE and SQUARE_SIZE_PIX
# compute SQUARE_SIZE constant

# create PsychoPy window, computing it size in pixels from the two constants

# create a square (Rect stimulus) with size of SQUARE_SIZE and position computed
#   via grid_to_win function from a pair of grid coordinates
```

```
# draw square and flip the window

# wait for any key press (just so that window stays on the screen)

# close PsychoPy window
```

Put your code into *exercise02.py*.

13.5 Adding the snake

Now, let us think about how can we represent a snake. It consist of one or more segments. The first one is its head, while the last one is its tail[4]. Thus, we can think about it as a list of individual segments.

Now let us consider an individual segment. We need to 1) keep track of its location in *grid* coordinate system and 2) have its visual representation — a square, just like the one you create during the previous exercise — positioned in *window* coordinate system. Good news is, you already have a function that maps the former on the latter, so as long as you know where the segment is on the grid, placing its square within the window is easy. As each segment has *two* pieces of information associated with it, it makes sense to represent it as a dictionary with two keys: `"pos"` (tuple of x and y in grid coordinates) and `"visuals"` (the square).

To keep code nicely compartmentalized and isolated, add a new function to *utilities.py* called `create_snake_segment`. It should take three arguments:

1. a variable with PsychoPy window, which you need to create an square visual.
2. a tuple `(x, y)` with segment position within the grid.
3. a square size, which you need for `map_grid_to_win` function.

It should return a dictionary with `"pos"` and `"visuals"` field, as we agreed upon above.

In the main code, create a variable `snake` that is a list with a single segment in the center of the screen (you need to compute it from `GRID_SIZE` using floor division operator //). Instead of drawing a single square one call at a time, as in the previous exercise, you will need to draw all segments of the snake using a *for* loop[5]. Here is the outline:

[4] A single segment snake is a special case, as its head is also its tail!

[5] You don't *really* need it for a single segment snake we have now but there is no other way to do it later, so let us do it properly from the start

```
# import all libraries and modules you need

# import map_grid_to_win and create_snake_segment functions from utilities.py file

# define constants GRID_SIZE and SQUARE_SIZE_PIX
# compute SQUARE_SIZE constant

# create PsychoPy window, computing it size in pixels from the two constants

# create the snake variable as a list with a single segment at the center of the scree


# draw all snake segments using for loop
# flip the window

# wait for any key press (just so that window stays on the screen)

# close PsychoPy window
```

Put function *create_snake_segment* into *utilities.py* and your modified code into *exercise03.py*.

13.6 Adding main game loop

Our current game is not *very* dynamic: it draws the (very small) snake once and waits for a key press before closing the window. Let us add the necessary scaffolding of a main game loop. The game should run while variable `gameover` is `False`. Inside the main loop, on each iteration you should draw the snake (and flip the window after that) and check whether participant press `"escape"` key via event.getKeys(). If they did, `gameover` should become `True`.

The code outline is

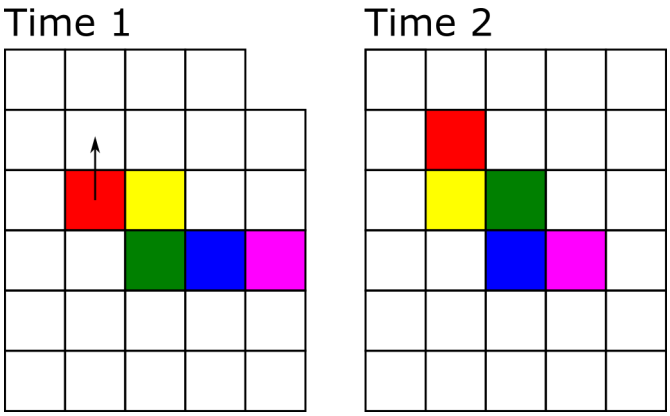
```
# import all libraries and modules you need

# import map_grid_to_win and create_snake_segment functions from utilities.py file

# define constants GRID_SIZE and SQUARE_SIZE_PIX
# compute SQUARE_SIZE constant

# create PsychoPy window, computing it size in pixels from the two constants

# create the snake variable as a list with a single segment at the center of the scree
```

```
# set gameover to False

# while not gameover:
#     draw all snake segments using for loop
#     flip the window

#     if escape key was pressed:
#         set gameover to True

# close PsychoPy window
```

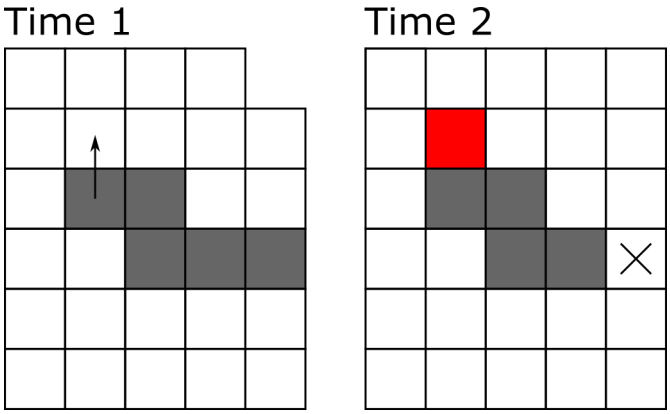
Put your code into *exercise04.py*.

13.7 Get a move on!

Now we need to understand how we will move the snake given that it consist of many segments. Assume that we have a four segment snake that moves up, as in the picture below.



Technically, we need to move each segment to a new position. The very first "head" segment moves to the position above the snake. The second segment moves to where the head was before. The third moves into the previous position of the forth one, etc. We *can* implement movement like that but instead we will utilize the fact that, unless colored as in the figure above, all segments look identical. Look at the uniformly colored snake below. We can "move" it by adding a new segment at a new location (growing a new head at the bity end of the snake, marked as red) and clipping off the last tail segment (marked by the cross), so that the previously penultimate segment becomes the tail. The rest can stay where they are, saving us a lot of hustle when the snake is long!



In the program, we list.insert(index, value)[6] the new head segment at index 0 and we remove the tail via list.pop().

To keep things neat and tidy, we will implement two functions: `grow_snake()` and `trim_snake()`. Why not combine both actions into a single function? Strategic thinking! Once we add apples to the game, the snake will grow after consuming them by not shedding the tail. Thus, separating these two functions now will simplify our lives later on[7].

The `grow_snake()` function should take the following parameters:

1. parameter for the snake itself, which we will modify. To minimize the potential confusion, do not call it `snake` but something (slightly) different, e.g., `the_snake`. This way it will be easier to remember that this is a parameter, not the global vairable.
2. a variable with PsychoPy window, which you need to create a new head segment.
3. a tuple `(x, y)` with *direction of movement* (call it `dxy` for change in xy coordinates). I.e., it will be `(-1, 0)` for leftward moving snake, `(1, 0)` for right moving, `(-1, 0)` for up, and `(1, 0)` for down.
4. a square size, which you need for `create_snake_segment` function.

Inside the function, you compute the coordinates for the new head segment based on the position of the current head (the very first element of the snake) and the direction of motion, (the third parameter `dxy`,) create a new snake segment at that location (you wrote a function for that!) and insert it before all other elements.

[6] We could also have used `dqueue` class from *collections* library instead of the `list`. It is, essentially, a list that allows appending and poping from the left as well.

[7] To be honest, I have initially implemented it as a single `move_snake()` function , wrote things up to eating apples, realized the problem, returned and rewrote the notes. So, it is a hindsight type of strategic thinking.

Somewhat counterintuitively, this function does not need to return anything because the snake, which we are passing to it as a first parameter, is a *mutable* list. Therefore, it is passed *by reference* and any changes that we apply to it inside the function will affect the global variable itself[8].

The `trim_snake()` function is even simpler as it takes the snake as the only parameter and trims its tail. Again, because the snake is a mutable list, you do not need to return anything.

To test these functions, let us create a three-segment long snake at location `(10,9)`, `(11, 9)`, and `(12, 9)` and move it (grow, then trim) upwards every time the player presses `"space"` button. This does not look like a real game but will make it easier to debug, as you can put a break-point to check computation step-by-step. Also note that, eventually, the snake will crawl out of the window but we will worry about this later. Here is the code outline:

```
# import all libraries and modules you need

# import all functions you need from from utilities.py file

# define constants GRID_SIZE and SQUARE_SIZE_PIX
# compute SQUARE_SIZE constant

# create PsychoPy window, computing it size in pixels from the two constants

# create the snake variable as a list with three segments

# set gameover to False

# while not gameover:
#     draw all snake segments using for loop
#     flip the window

#     if escape key was pressed:
#         set gameover to True
#     else if the key was "space":
#         move snake up
#
#
# close PsychoPy window
```

Put functions *grow_snake* and *trim_snake* into *utilities.py* and your modified code into *exercise05.py*.

[8] Re-read about mutable objects, if you forgot how it works.

13.8 Self-motion

Our snake should move by itself, not when the player presses *space* button. For this, we can call `grow_snake()` and `trim_snake()` functions on every iteration of the main game loop. However, by default, when you call `win.flip()` it will synchronize your loop with the refresh rate of the screen (typically, 60 Hz). This means that we would call these functions 60 times per second or, to put it differently, the snake will move 60 squares per second. This is waaaay too fast, given that our original grid size was just 30×20 squares. To appreciate just how fast this is, remove `if key was space:...` conditional statement, and call grow/trim snake functions on every iteration and see the snake fly off the screen.

Instead, we should decide on snake's speed, e.g., 4 squares per second, and define it as a new constant `SNAKE_SPEED`[9]. From that we can compute a new constant (`SNAKE_TIME_INTERVAL`) that expresses the time interval between calls of grow/trim functions, so that it is called `SNAKE_SPEED` times per second[10]. For 4 squares/second, that time interval will be 0.25 seconds. Do not hard code it, compute it from `SNAKE_SPEED`!

Now, we just need to use a clock or a countdown timer[11] We create it just before the main loop. Inside the loop, we check whether the elapsed time is equal to or greater than `SNAKE_TIME_INTERVAL`. If it is, move the snake and reset the timer. Try setting different speed and see whether the snake moves consistently.

```
# import all libraries and modules you need

# import all functions you need from from utilities.py file

# define constants GRID_SIZE and SQUARE_SIZE_PIX
# compute SQUARE_SIZE constant
# define SNAKE_SPEED and compute SNAKE_TIME_INTERVAL

# create PsychoPy window, computing it size in pixels from the two constants

# create the snake variable as a list with three segments

# set gameover to False
```

[9] Note that speed does not need to be an integer number of squares per second. It can move at 1.5 squares per second, so 3 squares every two seconds.

[10] Why two constants that express the same information? We could define only the `SNAKE_TIME_INTERVAL` but for a human it makes it harder to understand just how fast the movement will be. In cases like these, I prefer to have two constants, one human-oriented, another computer-oriented. Remember, it is not just about writing a working code, it is about writing a code that is easy for a human to understand.

[11] They work the same they, it is just a question of whether you start from zero and check whether time is over `SNAKE_TIME_INTERVAL` (clock) or you start at `SNAKE_TIME_INTERVAL` and check whether the time ran out/is already negative (timer).

```
# create a clock or a countdown timer
# while not gameover:
#     draw all snake segments using for loop
#     flip the window

#     if escape key was pressed:
#         set gameover to True
#
#     Check whether elapsed time is greater than SNAKE_TIME_INTERVAL,
#         move snake up and reset clock or countdown timer, if that is the case
#
#
# close PsychoPy window
```

Put your code into *exercise06.py*.

13.9 Describing direction using words

In our current design, we describe direction as a tuple `(dx, dy)`. Let us change it, so that it is described using words `"up"`, `"down"`, `"left"`, and `"right"`. This is not strictly necessary but will make our lives somewhat easier later on when we add steering controls. More importantly, it will serve a didactic purpose as well, showing how you can use dictionaries to translate values from one representation to another.

Thus, let us create a new string variable `direction` and set it to `"up"` (or any other direction you like). We will keep the `grow_snake()` function as is and will translate individual strings to pairs of `(dx, dy)` values. E.g., `"up"` corresponds to `(0, -1)`, `"right"` to `(1, 0)`, etc.

We can implement this translation via if-elif conditional statements:

```
if dir == "up":
  dxy = (0, -1)
elif dir == "right":
  dxy = (1, 0)
  ...
```

However, this approach introduces a lot of repetitive code and does not scale particularly well. Instead, we can use a dictionary (let us call it `DXY` as it is another constant) with `"up"`, `"right"`, etc. string as keys and tuples `(0, -1)`, `(1, 0)`, etc. as values. This way, we can use current value of `dir` variable as a key to get the `(dx, dy)` pair from `DXY` dictionary. Note that you should do it in the function call, **do not** create a temporary variable as in the if-elif examples above.

```
# import all libraries and modules you need

# import all functions you need from from utilities.py file

# define constants GRID_SIZE and SQUARE_SIZE_PIX
# compute SQUARE_SIZE constant
# define SNAKE_SPEED and compute SNAKE_TIME_INTERVAL

# define DXY dictionary with (dx, dy) entries for each "up", "down", "right",
#     and "left" keys

# create PsychoPy window, computing it size in pixels from the two constants

# create the snake variable as a list with three segments

# initialize direction variable to one of the four directions
# set gameover to False
# create a clock or a countdown timer
# while not gameover:
#     draw all snake segments using for loop
#     flip the window

#     if escape key was pressed:
#         set gameover to True
#
#     Check whether elapsed time is greater than SNAKE_TIME_INTERVAL,
#         move snake translating direction variable into (dx, dy) pair from DXY
#         reset clock or countdown timer, if that is the case
#
#
# close PsychoPy window
```

Put your code into *exercise07.py*.

13.10 It is all about control

Playing the game would be more fun, if we could steer the snake! If the player presses *right arrow* key, the snake should turn *clockwise*. Conversely, *left arrow* key, should turn the snake *counterclockwise*. We need to figure out two things. First, how to determine a new direction of motion given the current one and the key that was pressed. Second, we must decide when and how to change the value of `direction` variable. Let us tackle these problems one at a time.

Determining the new direction of motion is fairly straightforward. If current

is `"up"` and key was *right* (*clockwise* rotation), the new direction should be `"right"`. If current is `"down"` and key was *left*, the new direction is again `"right"`, etc. You could implement it as a bunch of `if-elif` statements or, better still, use the dictionary look up approach we implemented in the previous exercise. Here, you need a nested dictionary (dictionary inside a dictionary) `NEW_DIRECTION[key][direction]`. The first level has two keys `"left"` and `"right"` (so, effectively, counterclockwise and clockwise rotation) that selects which translation should be used and the second level is the dictionary that translates current direction into the new direction of motion. E.g., if current direction is `"down"` and key was `"right"`, `NEW_DIRECTION["right"]["down"]` should be `"left"` (rotating clockwise from `"down"` gets us to `"left"`). You know how define a simple dictionary. Good news, defining nested dictionaries follows the same rules, so should be straightforward.

Now let us think about when and how should we change value of `direction` variable. The simplest approach would be to change it as soon as the player presses the key. However, because our snake does not move on every frame this could lead to some odd behavior. Imagine that our game is on "easy" mode, so that the snake moves very slowly (one square per second). In this case, the player could easily press *left* twice during that second that would make snake move backwards, because direction was changed by 180°. Snakes, at least our snake, cannot do this. Thus, we need a temporary variable, let us call it `new_direction`, which we will set every time the player presses the key but whose value will be transferred to `direction` only when it is time to move the snake. We will compute it from the current `direction` and the key pressed. This way, even when the player presses *left* key several times, the snake would still turn only once because we compute the each turn using the same `direction` value but not the changing `new_direction` variable. This also means that players can "change their mind", as the last key press before the snake moves will determine the direction of motion.

```
# import all libraries and modules you need

# import all functions you need from from utilities.py file

# define constants GRID_SIZE and SQUARE_SIZE_PIX
# compute SQUARE_SIZE constant
# define SNAKE_SPEED and compute SNAKE_TIME_INTERVAL

# define DXY dictionary with (dx, dy) entries for each "up", "down", "right",
#     and "left" keys

# define NEW_DIRECTION nested dictionary

# create PsychoPy window, computing it size in pixels from the two constants
```

```
# create the snake variable as a list with three segments

# initialize direction variable to one of the four directions
# initialize new_direction(!!!!) to the same value as direction, why do you need to do

# set gameover to False
# create a clock or a countdown timer
# while not gameover:
#     draw all snake segments using for loop
#     flip the window

#     if escape key was pressed:
#         set gameover to True
#     if key is left or right:
#         set new_direction based on direction and pressed key using NEW_DIRECTION dic
#
#     Check whether elapsed time is greater than SNAKE_TIME_INTERVAL,
#         set direction to new_direction
#         move snake translating direction variable into (dx, dy) pair from DXY
#         reset clock or countdown timer, if that is the case
#
#
# close PsychoPy window
```

Put your code into *exercise08.py*.

13.11 Turning the hard way

Let us implement the same "figure out new direction" code in a more complicated way. The purpose of the exercise is to challenge you, show you new methods of the list, and to demonstrate how you can think about a change of the direction as moving through the list. We won't use this in the main code, so implement it as a function `compute_new_direction(current_direction, pressed_key)`, which will take two parameters (current direction and pressed key) and will return the new direction of rotation. I strongly recommend writing and debugging code in Jupyter Notebook first and copying it to *utilities.py* at the very end. Also, do not implement it as a function to begin with. Define parameters as two variable with some preset values and debug the rest of the code.

Here is the idea. Imagine that you have a list `["left", "up", "right", "down"]`. For this list, rotation clockwise would correspond to moving through the list to the right (assuming that you jump to the beginning once you move

past the last item). Conversely, rotation counterclockwise corresponds to moving to the left (again, assuming that you jump to the end of the list, once you went past the first item). As you see, rotation is expressed as a very intuitive "motion through the list".

For the actual implementation, first, define a constant list in the order I've described. Next, you need to identify the location of the current direction within the list using index() method. Then, you need to figure out whether you increase or decrease that index (to move to the right or to the left), based on `pressed_key` parameter (you can use dictionary approach or a conditional assignment). Finally, you need to control for range, so that index of `-1` becomes `3` (you went too far to the left) and index of `4` should become `0`. The most elegant way to do this, is using `%` modulus operation. Hint, `4 % 4` is `0`. What about `1 % 4`, `0 % 4`, or even `-1 % 4`? Check it by hand to get an idea of what I am hinting at. And, of course, do not use `4` for division, use the length of the list, as it determines the range of values.

Put function compute_new_direction into *utilities.py*.

13.12 To be continued...

We covered a lot of ground today. Next time, we will continue by adding the possibility of killing yourself but also of growing into a longer and cooler snake by eating apples.

Chapter 14

Snake game, part 2

Our plan for today is to continue developing the snake game. Our original plan was as follows and we got up to #4.

1. Create boilerplate code to initialize PsychoPy.
2. Figure out how to place a square. We need this because our snake is made of square segments and lives on a rectangular grid made of squares.
3. Create a single segment stationary snake.
4. Make the snake move assuming a rectangular grid.
5. Implement "died by hitting a wall".
6. Add apples and make the snake grow.
7. Add check for biting itself.
8. Add bells-and-whistles to make game look awesome.

As before, create a new folder for this seminar, copy the latest version of *utilities.py* to it and base your further development code on the latest exercise from the previous seminar (should be *exercise08.py*).

14.1 Hitting the wall

The last thing that we implemented during the previous seminar was our ability to control the snake. However, you could steer it off the screen and make it go through itself. Let us fix the former!

Develop a new function `hit_the_wall` and put it into *utilities.py*. The function should take `snake` variable as a first parameter and check whether the head of the snake (which segment is it?) is still within the grid that we defined (how do you check for that?). The function should return a logical value. `True`, if the head of the snake is outside of the grid limits (so, it is true that the snake

hit the wall). `False`, if it is still inside. I mentioned that you should definitely pass the `snake` as an obvious parameter of the function but do you need other parameters? If yes, which ones? Document the function!

Test the function by adding a new condition inside the main game loop. Check whether the snake `hit_the_wall()` and, if that is the case, the game should be over. Think about the optimal place where to check for this. You could do it on every iteration but there is a more logical place for it, inside that main loop. Where is it?

Put function *hit_the_wall* into *utilities.py* and your updated code into *exercise01.py*.

14.2 Is this the snake?

In the next section, we will be adding apples to the game. The catch is that these apples should appear at a location that is *not* occupied by the snake. Otherwise, we would generate apples directly into snake's stomach. Practical for the snake but defeats the purpose of the game. To rephrase this problem, we need a function that checks whether a particular grid location is occupied by the snake.

Develop a new function `is_inside_the_snake` that takes the snake as a first parameter and a tuple with a grid location as a second parameter and returns a logical value whether that grid location is occupied by the snake (`True`) or not (`False`). Document the function!

My approach would be to work on this function in a Jupyter Notebook first. To debug it, create a snake by hand hardcoding the list of dictionaries. Note that you only need `pos` part of the segment's dictionary for this function, so can happily ignore the `visuals` key. Once you feel the function works, copy it to *utilities.py*.

Put function *is_inside_the_snake* into *utilities.py*.

14.3 An inedible apple

Let us add that highly desirable fruit: the apple! We will represent it as a *differently* colored square and snake won't be able to eat it yet. I assume that you used green color for the snake, so apples would yellow, red, or orange (but feel free to pick any color of your fancy). Just like a snake segment, an apple is characterized by its position on the grid and by its visuals, so you can use the same dictionary structure as for the snake's segment.

You need to write a new function `create_apple` that will be fairly similar to `create_snake_segment()` function you developed the last time. It should also

return a dictionary with `pos` and `visuals` field. However, instead of taking a predefined grid location, it should generate one randomly within the grid making sure it is *not* on the snake. The `is_inside_the_snake` function will help you to ensure the latter. Think about parameters that you need for this function. Remember to document it!

In the main loop, create a new variable `apple` and initialize it via `create_apple()` function. Think about where and when do you need to initialize it. Also, you need to drawi the apple in the main loop. Again, think about where should you do it.

Put function *create_apple* into *utilities.py* and your updated code into *exercise02.py*.

14.4 Eating an apple

Apples exist for snakes to eat them! Let us add this functionality. The general idea is very simple. If the *head* of the snake moves on to the grid location with an apple, you should not trim its tail. See how useful it was to split growing and trimming into two separate functions? Told you, strategic thinking!

You need to add a conditional statement that if the snake's head *is* on the apple, you should generate a new one the same way as you generated it initially (add the next apple). What should you do, if there is no apple at that location?

Also, if you haven't already done that, modify you code so that you start with a single segment snake.

Put your updated code into *exercise03.py*.

14.5 Eating yourself

Once our snake grows beyond four segments, it has an opportunity to bite itself[1]. For this, we need to check that, *after* the snake moved, its head is *not* at the same location as one of the segments of its body. Create a new function `snake_bit_itself()` that takes the snake as a single parameter and returns `True` or `False` based on whether that is the case. The function is very similar but not identical to `is_inside_the_snake()` function you implemented earlier. What is the critical difference and why cannot you simply reuse that function?

Once you implemented `snake_bit_itself()` function, you should check for that eventually after it moved and, if that is the case, the game should be over.

Put function *snake_bit_itself* into *utilities.py* and your updated code into *exercise04.py*.

[1] Why at least five? Draw it on the grid and figure out whether it can do so with four.

14.6 Bells and whistles: score

Now that we have a fully functional game, we can start adding non-essential but nice features to it. The first one will be the score that is equal to the length of the snake. It should read *Score: XXX*. Place it at the top of the window. You will need to use TextStim class for this. Think about when to create the text stimulus, when to draw it, and how and when to update the text.

Put your updated code into *exercise05.py*.

14.7 Bells and whistels: three lives

Let us give the player three attempts to achieve the top score. They have three lives, every time the snake dies, the game *but not the score* resets: A single segment snake appears at the center and a new random apple appears elsewhere (where should you put the code to create them?). Once the snake dies three times, the game is over. Think how you can implement this three repetitions.

The score should be cumulative, so at the beginning of round two it should be equal to the final score of round one plus current length of the snake (1 at the very beginning). Think how you can achieve this. *Another important point*: now you have two nested loop, one is for the game, one is for the round. When the snake dies, the round is over and, if you run out of lives, the game as well. When the player presses *escape* both round **and** the game are over. Think about how you can implement it.

Put your updated code into *exercise06.py*.

14.8 Bells and whistels: showing lives

Let us not just repeat the game three times but show the player how many lives they still have. Download the heart.png[2] and use it show remaining lives at the top-left corner of the screen: three hearts in round one, two hearts in round two, and just a single heart in round three. You will need to use (ImageStim)[https://www.psychopy.org/api/visual/imagestim.html#psychopy.visual.ImageStim] for this. Think about the size of images and their location.

Put your updated code into *exercise07.py*.

[2]This image was downloaded from openclipart.org and was created by cliparteles

14.9 Bells and whistles: difficulty

At the moment, the difficulty of the game, the speed with which the snake moves, is fixed and the player has no way of choosing it. Let us create dialog that appears *before* we create the window and start the game that will allow the player to choose between *easy*, *normal*, and *difficult*[3]. I leave it up to you to decide which snake speeds correspond to each difficulty (and you can have more than three options, if you want).

To create and run the dialog, use Dlg class from giu module of PsychoPy. Your challenge for today is to figure out how to use it based on the manual along. Take a look at the example and experiment with in a separate file.

Put your updated code into *exercise08.py*.

14.10 Bells and whistles: sounds

Download game-over-arcade.wav[4] and 8-bit-game-over-sound.wav[5]. Use the former whenever the snake dies and use the latter when the player runs out of lives. You will need to use Sound class from sound module of PsychoPy.

Put your updated code into *exercise09.py*.

14.11 Bells and whistles: blinking game over message

Once the game is over, show a blinking "Game Over" message *superimposed* over the final static game screen. Thus, you need to draw all the game objects and messages (but without moving the snake) plus you show a text message that is on for 0.5 second and off for 0.5 seconds until the player presses *Space* button. Hint, it should be a separate loop after the main game loop over rounds and clock/timers have definitely something to do with it.

Put your updated code into *exercise10.py*.

14.12 Next stop: classes and objects

Nice game! Next time you will learn about classes and objects and you will rewrite the Snake Game using them. This way, we can concentrate on figuring

[3]Or, if you played Doom, between *I'm Too Young To Die*, *Hey, Not Too Rough*, *Hurt Me Plenty*, *Ultra-Violence*, and *Nightmare*.

[4]Downloaded from freesound.org and created by myfox14

[5]Also downloaded from freesound.org and created by EVRetro

out classes, as you already know how the game works.

Chapter 15

Snake game: object-oriented programming

We will not be programming a new game today. Rather, you will learn about object-oriented approach in Python and will use it to simplify your code of the Snake game that you already programmed. Yes, it would be more fun to program a new game but this way you get to concentrate on the concepts rather than on the game logic.

15.1 Object-oriented programming

The core idea is in the name: Instead of having variables/data and functions separately, you combine them in an object that has attrbutes/properties (its own variables) and methods (functions). This approach uses our natural tendency to perceive the world as a collection of interacting objects and has several advantages that I will discuss below.

15.1.1 Classes and objects (instances of classes)

Before we continue, I need to make an important distinction between *classes* and *objects*. A *class* is a "blue print" that describes properties and behavior (methods) of objects of that class. This "blue print" is used to create an *instance* of that class, which is called an *object*. For example, Homo sapiens is a *class* that describes species that have certain properties, such as height, and can do certain things, such as running. However, Homo sapiens as a class only has a

concept of height but no specific height itself. E.g., you cannot ask "What is height of Homo sapiens?" only what is an average (mean, median, etc.) height of individuals of that class. Similarly, you cannot say "Run, Homo sapiens! Run!" as abstract concepts have trouble with real actions like that. Instead, it is Alexander Pastukhov who is an *instance* of Homo sapiens class with a specific height and a specific (not particularly good) ability to run. Other instances of Homo sapiens (other people) will have different height and a different (typically better) ability to run. Thus, class describes what kind of properties and methods objects have. This means that whenever you meet a Homo sapien, you could be sure that they have height. However, individual objects have different values for this properties and so calling their methods may result in different outcomes.

Another, a more applied, example would be your use of `ImageStim` *class* to create multiple *instances* of front side of a card in "Memory" game. Again, the *class* defines properties (`image`, `pos`, `size`, etc.) and methods (e.g., method `draw()`) that individual *objects* will have. You created these objects to serve as front side of cards. You set *different* values for same properties (`image`, `pos`) and that ensured that when you call their method `draw()`, each card was drawn at its own location and with it own image.

15.1.2 Encapsulation

Putting all the data (properties) and behavior (methods) inside the class simplifies programming by ensuring that all relevant information can be found in its definition. Thus, you have a single place that should hold *everything* that defines object's behavior. Contrast this with our approach in previous two seminars where snake data (e.g., variable `snake`) was defined at one place and functions that used and altered it (e.g., `snake_bit_itself()` or `grow_snake()`) were defined elsewhere. Moreover, we had to resort to using *snake* or *apple* in function names just to remind ourselves that they belong to the snake or an apple. This also necessitated creating functions with multiple arguments that ensured that all information is available within each function. And, we modified `snake` inside the function creating further uncertainty about when and where it can be changed. Today, you will see how encapsulating everything into classes turns this mess into a simpler and easier-to-understand code.

15.2 Inheritance / Generalization

In object-oriented programming, a class can be derived from some other *ancestor* class and thus *inherit* its properties and methods. Moreover, several classes can be derived from a single ancestor producing a mix of unique and shared functionality. This means that instead of rewriting the same code for each class, you can define a common code in an ancestor class and focus on differences or additional methods and properties in descendants.

Using the Homo sapiens example from above. Humans, chimpanzees and gorillas are all different species but we share a common ancestor. Hence, we are different in many respect, yet, you could think about all of us as "apes" that have common properties such as binocular trichromatic vision. On other words, if you are interested in color vision, you do not care what specific species you are looking it, as all apes are the same in that respect. Or, you can move further down the evolution tree and think about us as "mammals" that, again, have common properties and behavior, such as thermoregulation and lactation. Again, if you are interested *only* in whether an animal has thermoregulation, knowing that it is a mammal is enough.

Similarly, in PsychoPy various visual stimuli that we used (`ImageStim`, `TextStim`, `Rect`) have same properties (e.g., `pos`, `size`, etc.) and methods (most notably, `draw()`). This is because they are all descendants from a common ancestor `BaseVisualStim` that defines their common properties and methods[1]. This means that you can assume that *any* visual stimulus (as long as it descends from `BaseVisualStim`) will have `size`, `pos`, `ori` and can be drawn. This, in turn, means that you can have a list of various PsychoPy visual stimuli and move or draw all of them in a single loop without thinking which *specific* visual stimulus you are moving or drawing. Also note that you cannot assume these same proporties of *sound* stimuli because they are *not* descendants of `BaseVisualStim` but of `_SoundBase` class.

Where is another way of achieving common behavior (generalization) in Python without inheritance. It is called "duck typing"[2] and it will be the topic of a different seminar.

15.3 Polymorphism

As you've learned in the previous section, inheritance allows different descendants to share common properties and behavior, so that in certain cases you can view them as being equivalent to an ancestor. E.g., any visual stimulus (a descendant of `BaseVisualStim` class) can be drawn, so you just call its `draw()` method. However, it is clear that these different stimuli implement drawing *differently*, as the `Rect` stimulus looks different from the `ImageStim` or `TextStim`. This is called "polymorphism" and the idea is to keep the common interface (same `draw()` call) while abstracting away the actual implementation. This allows you to think about what you want an object to do (or what to do with an object), instead of thinking how exactly it is implemented.

[1] `BaseVisualStim` does not actually define `draw()` method, only that it must be present.
[2] Yes, it is really called "duck typing".

15.4 A minimal class example

Enough of the theory, let us see how classes are implemented in Python. Here is a very simple class that has nothing but the *constructor* `__init__()` method, which is called whenever a new object (class instance) is created, and a single attribute / property `total`.

```
class Accumulator:
    """
    Simple class that accumulates (sums up) values.

    Properties
    ----------
    total : float
        Total accumulated value
    """

    def __init__(self):
        """
        Constructor, initializes the total value to zero.
        """
        self.total = 0

# here we create an object number_sum, which is an instance of class Accumulator.
number_sum = Accumulator()
```

Let's go through it line by line. First line `class Accumulator:` shows that this is a declaration of a `class` whose name is `Accumulator`. Note that the first letter is capitalized. This is not required per se, so Python police won't be knocking on your door if you write it all in lower or upper case. However, the general recommendation is that **class** names are written using `UpperCaseCamelCase` whereas **object** (instances of the class) names are written using `lower_case_snake_case`. This makes distinguishing between classes and objects easier, so you should follow this convention.

The definition of the class are the remaining *indented* lines. As with functions or loops, it is the indentation that defines what is inside and what is outside of the class. The only method we defined is `def __init__(self):`. This is a *special* method[3] that is called when an object (instance of the class) is created. This allows you to initialize the object based on parameters that were passed to this function (if any). You do not call this function directly, rather it is called whenever an object is created, *e.g.* `number_sum = Accumulator()` (last line). Also, it does not return any value explicitly via `return`. Instead, `self` (the very first parameter, more on it below) is returned automatically.

[3]There are more special methods that you will learn about later, they all follow `__methodname__()` convention.

All class methods (apart from special cases we currently do not concern ourselves with) must have one special first parameter that is *the object* itself. By convention it is called `self`[4]. It is passed to the method automatically, so whenever you write `square.draw()` (no explicit parameters written in the function call), the actual method still receives one parameter that is the *reference* to the `square` variable whose method you called. Inside a method, you use this variable to refer to the object itself.

Let us go back to the constructor `__init()__` to see how you can use `self`. Here, we add a new *persistent* attribute/property to the object and assign a value to it: `self.total = 0`. It is *persistent*, because even though we created it inside the method, the mutable object is passed by reference and, therefore, we assigned it to the object itself. Now you can use this property either from inside `self.total` or from outside `number_sum.total`. You can think of properties as being similar to field/value pairs in the dictionary we used during previous seminar but for syntax: `object.property` versus `dictonary["field"]`[5]. Technically, you can create new properties in any method or even from outside (e.g., nothing prevents you from writing `number_sum.color = "red"`). However, this makes understanding the code much harder, so the general recommendation is to create *all* properties inside the constructor `__init__()` method, even if this means assigning `None` to them[6].

15.5 add method

Let us add a method that adds 1 to the `total` property.

```
class Accumulator:
    ... # I am skipping all previous code here

    def add(self):
        """
        Add 1 to total
        """
        self.total += 1
```

It has first special argument `self` that is the object itself and we simply add 1 to its `total` property. Again, remember that `self` is passed automatically whenever you call the method, meaning that an actual call looks like `number_sum.add()`.

[4] Again, you can use any name but that will surely confuse everyone.

[5] This is actually how all properties and methods are stored, in a `__dict__` attribute, so you can write `number_sum.__dict__["total"]` to get it.

[6] If you use a linter, it will complain whenever it sees a property not defined in the constructor

Create a Jupyter notebook (you will need to submit it as part of the assignment) and copy-paste the code for `Accumulator` class, including the `.add()` method. Create **two** objects, call them `counter1` and `counter2`. Call `.add()` method twice for `counter2` and thrice for `counter1` (bonus: do it using `for` loop). What is the value of the `.total` property of each object? Check it by printing it out.

Copy-paste and test `Accumulator` class code in a Jupiter notebook.

15.6 Flexible accumulator with a `subtract` method

Now lets us create a new class that is a *descendant* of the `Accumulator`. We will call it `FlexibleAccumulator` as it will allow you to also *subtract* from the total count. You specify ancestors (could be more than one!) in round brackets after the class name

```
class FlexibleAccumulator(Accumulator):
    pass # You must have at least one non-empty line, and pass means "do nothing"
```

Now you have a new class that is a descendant of `Accumulator` but, so far, is a perfect copy of it. Add `subtract` method to the class. It should subtract `1` from the `.total` property (don't forget to *document* it!). Check that it works. Create one instance of `Accumulator` and another one of `FlexibleAccumulator` class and check that you can call `add()` on both of them but `subtract()` only for the latter.

Add `subtract` method to the `FlexibleAccumulator` class in a Jupiter notebook. Add testing.

15.7 Method arguments

Now, create a new class `SuperFlexibleAccumulator` that will be able to both `add()` and `subtract()` *arbitrary* value! Think about which class it should inherit from. Redefine both `.add()` and `.subtract()` method in that new class by adding `value` argument to both method and add/subtract this value rather than `1`. Note that now you have *two* arguments in each method `(self, value)` but when you call you only need to pass the latter (again, `self` is passed automatically). Don't forget to document `value` argument (but you do not need to document `self` as its meaning is fixed).

Create `SuperFlexibleAccumulator` class and define super flexible `add` and `subtract` methods that have `value` parameter (in a Jupiter notebook). Test them!

15.8 Constructor arguments

Although constructor `__init(...)__` is special, it is still a method. Thus, you can pass arguments to it just like you did it for other methods. You pass these arguments when you create an object, so in our case, you put it inside the bracket for `counter = SuperFlexibleAccumulator(...)`.

Modify the code so that you pass the initial value that total is set to, instead of zero. ::: {.infobox .program} Add `initial_value` parameter to the constructor of the `SuperFlexibleAccumulator` class in a Jupiter notebook. Test it! :::

15.9 Calling methods from other methods

You can call a function or object's method at any point of time, so, logically, you can use methods inside methods. Let's modify our code, realizing that *subtracting* a value is like *adding a negative* value. Modify your code, so that `.subtract()` only negates the value before passing is to `.add()` for actual processing. Thus, `total` is modified *only* inside the `add()` method.

Modify `subtract()` method of `SuperFlexibleAccumulator` to utilize `add()` in a Jupiter notebook. Test it!

15.10 Local variables

Just like normal functions, you methods can have local variables. They are local (visible and accessible only from within the method) and are not persistent (their values do not survive between the calls). Conceptually, you separate variables that need to be persistent (retain their value the whole time object exists) as attributes/properties and temporary variables that are need only for the computation itself as local method variables. What would be value of property `.total` in this example:

```
class Accumulator:
    def __init__(self, initial):
        temp = initial * 2
        self.total = inital

counter = Accumulator(1)
```

What about in this case?

```
class Accumulator:
    def __init__(self, initial):
        temp = initial * 2
        self.total = temp

counter = Accumulator(1)
```

15.11 Refactoring the Snake game

Got the basic idea about using classes? Let us practice by partially refactoring the snake game. In principle, we can factor it into three classes / objects:

- a window class that will incorporate grid information and mapping
- an apple class
- a snake class

Here, we will cover the first two but I encourage you to think about how to convert snake-bits into a class as well. Before you begin, create a new folder (*snake2*, *snake-oop*, etc.) and copy your old code where.

15.12 GridWindow class

One of the most annoying things about our current code was the necessity to pass grid information (`GRID_SIZE`, `SQUARE_SIZE`, etc.) as well as window variable every time we needed to create a new segment, or an apple, or grow snake, etc. It would be much simpler, if our window object would also include information on the grid (grid and square sizes) as well as the function that maps grid-to-window coordinates (`map_grid_to_win()`). In this case, we would only need to pass the window variable that has all information that snake or apple function and classes will need.

We still want the window to behave like PsychoPy window, so our `GridWindow` will *inherit* from it. Create a new file *gridwindow.py* and create a new class as follows. We will discuss the specific bits below.

```
# 1. import

class GridWindow(Window):
    """
    Grid window, inherits from PsychoPy window and adds grid attributes and functions.
    """
```

```
    def __init__(self, grid_size, square_size_pix):
        """
        Parameters
        ----------
        grid_size : tuple
            Size of the grid
        square_size_pix : integer
            size of a single square in pixels
        """

        # 2. store grid_size as an attribute, so that later on you
        #     could use it as self.grid_size (from inside) and
        #     win.grid_size (from outside)

        # 3. compute square_size attribute

        # 4. call the constructor of the ancestor
        super().__init__(units="norm", size=(however-you-computed-it))

    def map_grid_to_win(self, ipos):
        """
        Converts grid coordinates to window coordinates in height units.

        Parameters
        -----------
        ipos : tuple
            (x, y) coordiantes on the grid

        Returns
        -----------
        tuple
            (norm_x, norm_y) coordinates in the window
        """

        # 5. code that computed the transformation but that uses
        #    class attribute for square size
```

Let us go through the code!

1. You do need to import some modules, which ones?
2. We pass the `grid_size` as a parameter to the constructor but we need to retain in for future computation. Hence, we should store it as a persistent attributed of the class. You can (should) use the same name but remember this is an attribute not a constant, so you should use `snake_lower_case`

for the name.

3. In the original code, you store the computed square size as a constant. Here, you store it as a persistent attribute but, otherwise, the idea is the same.
4. New and somewhat complicated bit! In the original code, we created a window and specified its size in pixels (computing it from the grid size and size of the square in pixels) and `"norm"` units. We still need to do this, so that the window is initialized properly. In the constructor we wrote the code that takes care of *new* bits but we need to ensure that the ancestor class does the *old* bits. For this, we need to call its constructor (`__init__()` method). But there is a catch, we *overwrote* it with a brand new `__init__()` method of our own, so we cannot just call it via `self.__init__()`[7]. Instead, Python has `super()` function that refers to the ancestor of the class. Hence, `super().__init__(your-arguments-inside)` initializes the window. Note that you need `super()` only if you *override* the original method, as we did with `__init__()`. E.g., you do not need it to call `flip()` method defined in the original class.
5. The code is essentially the same but you can use square size attribute, so no need to pass it as a separate argument.

And in the main code, you now create the window as

```
win = GridWindow(GRID_SIZE, SQUARE_SIZE_PIX)
```

15.13 Apple class

Create it in a separate *apples.py* file. The apple class has no ancestors and we just need to put all methods and variables together. Here are lists of attributes and methods.

Attributes:

- `win` : you should pass window variable to the constructor of an `Apple` class and store it as an attribute for later use when you (re)create the apple at random position.
- `pos` : same as the field in the original dictionary but now as an attribute.
- `visuals` : same as the field in the original dictionary but now as an attribute.

Methods:

[7] Well, we can write that but we will be calling out brand new `__init__()` again.

- `__init__(self, win)` should store window as attribute and create *empty* attributes `pos` and `visuals`.
- `reset(self, snake)` : modified version of `create_apple()` function but you use grid size and square size attributes of the `self.win` instead of passing them as arguments. It should overwrite any old values of `self.pos` and `self.visuals` instead of returning the dictionary.
- `draw()` : should draw the apple!

Now, you need to create an `apple` object *after* you create the window and `reset()` it whenever you need a new apple (at the beginning of each round and once the snake ate the apple).

15.14 Refactoring the code

Start by changing with window class to `GridWindow` without doing anything else. The code should "just work" as our `GridWindow` has all the functionality of PsychoPy `Window` that we relied upon. Next, go through the *snake* code (the apple has its own class, so we will deal with it later) to use window attributes instead of constants and simplify functions. E.g., `create_snake_segment(win, ipos, square_size)` does not need the third argument anymore, as you can get it from `win` itself. Similarly, you do not need it for `grow_snake()` function. Alter one function at a time and test your code.

Next, use `Apple` class for the apple and adjust the code accordingly using attributes and `draw()` method. Here, you have to do all alterations in one go, of course, as changes we introduced are breaking.

Remember, one step at a time, check your code before continuing. Put breakpoints and use debug console, if you are unsure about the changes.

Chapter 16

Moon lander game

Today we will create a moon lander game. You job is simple: land your ship on the pad but do not crash it! Here is a brief video of my implementation of the game

We will program it using object-oriented approach from the beginning by defining two classes: the `MoonLander` and the `LandingPad`. Here is the general outline of how we will proceed:

1. Create a basic PsychoPy window and main experimental loop.
2. Define a basic `MoonLander` class with a static image and add its drawing to the main loop.
3. Randomize position of the lander.
4. Add gravity pull.
5. Add vertical thruster that counter-act gravity.
6. Add horizontal thrusters, so you can maneuver around.
7. Define `LandingPad` class.
8. Learn about virtual attributes and implement them for both classes.
9. Implement landing / crashing checks.
10. Add more runs.
11. Limit the fuel.

This time most of the code will be in classes, so making versions of them will be quite cumbersome. Thus, submit only the final game. I would suggest calling files *main.py* (main script), *moonlander.py* (`MoonLander` class), and *landingpad.py* (`LandingPad` class).

16.1 Create window

Create a PsychoPy window 640 × 480 in size. Add a main game loop with `gameover` variable that can be exited by pressing *escape*.

Put your code into *main.py*.

16.2 Create MoonLander class

In *moonlander.py*, create a new `MoonLander` class. It should have an ImageStim as an attribute (that will be the visuals of the ship) created using ufo.png image. For the moment, place it at its default location at the center. Also, implement `draw()` method that should draw all visual elements of the lander (we have one now but there will be more later on).

Create an instance of `MoonLander` class in the main script and draw it in the main game loop. You should see a static picture of the ship at the center of the screen.

Create `MoonLander` class and use it in the main game loop.

16.3 Randomize position

Implement a new method `reset()` that resets the lander for the next round. At the moment, the only thing it should do is to randomize position of the image. Use range of -0.5..0.5 horizontally and 0.8..0.9 vertically. Call it in the constructor and test it in the main loop by calling it every time you press *space* button.

Add `reset()` method to `MoonLander` class and use it in the main game loop.

16.4 Gravity

Next we need to create a gravitational force that will pull the lander down. Create a constant `GRAVITY = 0.0001`[1] and create a new attribute `speed = [0, 0]` that is horizontal and vertical speed of the lander. Note that it should be reset to `(0, 0)` in the `reset()` method.

The position of the lander (`self.image.pos`) must adjusted based on the the speed on every frame. But before that, speed itself must be adjusted based on the forces from gravity and thrusters that act upon the lander. Create a

[1] The constant itself does not mean anything, I adjusted it to be reasonable for the image and window size that we are using.

new method `update()` where you first adjust *vertical* speed based on gravity alone (we will add the effect of thrusters later) and when adjust *vertical* position based on vertical speed (we will worry about the horizontal speed, once we start working on horizontal thrusters). Call `update()` method before the `draw()` in the main loop. Your lander should fall down at accelerated rate (you can play with `GRAVITY` constant to see how it changes the speed of falling). Once it is off the screen, press space and see it go again.

Update `MoonLander` class for the effect of gravity and use this in the main loop.

16.5 Vertical thurster

PsychoPy allows you to get key presses or, using hardware.keyboard to get both press and release time. Unfortunately, you get both only *after* the key was released. But in our game, the thursters must be active for as long as the player presses the key. Thus, we need to know whether a key is *currently* pressed, not that it was pressed and released at some time in the past. For this, we will use *pyglet* library (that is a backend used by PsychoPy) directly. First, in your *moonlander.py* add `import pyglet` and then include the following code inside the constructor of the class.

```
# setting up keyboard monitoring
self.key = pyglet.window.key
self.keyboard = self.key.KeyStateHandler()
win.winHandle.push_handlers(self.keyboard)
```

This installs a "handler" that monitors the state of the keyboard. Now, you can read out the state of, say, *down arrow* key as`self.keyboard[self.key.DOWN]` (`True` if pressed, `False` otherwise). We will use `DOWN` for the vertical thruster and `LEFT` and `RIGHT` for the horizontal ones.

Define a `VERTICAL_ACC` to be twice the gravity (but you can use some other number, of course) and update the `update()`[2], so that the total vertical acceleration is `VERTICAL_ACC + GRAVITY` if the the user is pressing *down* key (use `self.keyboard` and `self.key` to figure that out) but `GRAVITY` alone, if not.

Test you that vertical thruster works!

Update `MoonLander` class for the counter-effect of a vertical thruster.

16.6 Horizontal thursters

Now implement the same logic, computing acceleration, speed, and position but for horizontal thrusters (define `HORIZONTAL_ACC`, decide on its value yourself).

[2] Pun intended.

Remember, the *right* thruster pushes the lander to the *left* and vice versa! Assume that only one of the keys, *left* or *right*, can be pressed at a time. Test it by flying around!

Add horizontal thrusters to `MoonLander`.

16.7 Landing pad: visuals

The purpose of the game is to land on a landing pad. Let us add one! Create a new file *landing_pad.py* and a new class `LandingPad`. In the constructor, create a rectangle that is `0.5` units wide and is located at the bottom of the window but at a random position within the window horizontally. Pick the fill and line colors that you like. The only other method the class needs is `draw()`.

In the main code, create an object of class `LandingPad` and draw it in the main loop, along with the lander itself.

Create `LandingPad` class and use it in the main loop.

16.8 Computing edges of game objects

The aim of the game is a soft touchdown on a landing pad. For this, we need to know where the *top* of the landing pad is, as well as where the *bottom* of the lander is and where *left* and *right* limits of each object are. Let us think about *bottom* of the lander first, as the rest are very similar.

We do not have information about it *directly*. We have the vertical position of the lander in `self.image.pos[1]` (I assume here that the visuals attribute is called image) and its height in `self.image.size[1]`. From this, it is easy to compute the bottom edge (but remember that position is about the *center* of the rectangle). Accordingly, you could create a method called `bottom()` that would return the computed value when it is called (e.g., `lander.bottom()`).

Implement `bottom()` method for the `MoonLander` class.

16.9 Virtual attributes via getters and setters

Our approach of using `lander.bottom()` works but it is *semantically* inconsistent. Calling methods is about manipulating objects, e.g., drawing them, updating them, comparing them. However, `bottom` is, effectively, an *attribute* of an object, like its position or size, it is just where the bottom of it is. We *could* create and compute a `bottom` attribute inside the constructor, solving the problem of semantics, as now you could write `lander.bottom`. Note that as the lander moves all of its edge attributes (`bottom`, `left` and `right`) need

to be recomputed after every update. This is unavoidable but a *real* attribute approach still creates another problem: What if someone *changes* it? In that case, its value will not be correct, as `bottom` value depends on both position and size, so changing it without a corresponding change in those two attributes makes no sense! And it is really hard to decide whether change in the `bottom` should mean a change in position, or size, or, perhaps, both? Thus, ideally, it should be a read-only attribute.

For cases like these, Python has special *decorators*[3]: `@property` and `@<name>.setter`. The former one decorates a method that allows you to *get* an attribute's value and, typically, is called a "getter". The latter one, is for a "setter" method that *sets* a new value to an attribute. The idea is to isolate an actual attribute value from the outside influence. It is particularly helpful, if you need to control whether a new value of an attribute is a valid one, needs to converted, etc. For example, `color` attribute of the rectangle stimulus uses this approach, which is why it can take RGB triplets, hexadecimal codes, or plain color names as a value and set the color correctly.

Here is a sketch of how it could work but note that today, we will only use the *getter* bit. To have a virtual attribute for `color`, one typically creates an *internal* attribute with almost the same name, e.g., `_color` or `__color` (see below for the difference). The value is stored and read from that internal attribute by getter and setter methods:

```
class ExampleClass:

  def __init__(self):
    self.__color = "red"

  @property
  def color(self):
    """
    This is a getter method for virtual color
    attribute.
    """
    # Here, we simply return the value. But we could
    # compute it from some other attribute(s) instead.
    return self.__color

  @color.setter # not the most elegant syntax, IMHO
  def color(self, newvalue):
    """
    Note that the setter name has THE SAME name as the getter!
```

[3]These are functions that "decorate" you function and are called *before* your function is called. They are like gatekeeps or face control, so they can alter whether your function or how it is executed. They are very useful in certain scenarios and we will meet them again later. However, we will not talk about them in detail here.

```
    It sets a new value and does not return anything.
    """

    # Here, you can have checks, conversion,
    # additional changes to other attributes, etc.
    self.__color = newvalue


example = ExampleClass()

# get the value, note the lack of () after color
print(example.color)

# set the value
example.color = "blue"
```

Note that there is no *actual* attribute `color`, yet, our code works as if it does exist.

There is another twist to the story. If you only define the getter `@property` method but no setter method, your property is read-only[4]! And this exactly what we wanted. Turning out `bottom()` method into a getter of an attribute is as easy as adding the `@property` decorator to it. After that, we can use it as a read-only attribute `lander.bottom`. Do this and also create similar read-only attributes for `top`, `left`, and `right` of the pad and for `left`, and `right` of the lander class.

Implement virtual properties for `MoonLander` and `LandingPad` classes.

16.10 Access restrictions

In the example on getter/setter methods, I used `__color` name with *two* leading underscores. This is a Python way to make things (almost) private, that is, invisible from outside. If you copy-paste the class code from above and try to access the attribute directly via `example.__color`, you will get an error "'ExampleClass' object has no attribute '__color'". However, as I wrote, it is *almost* private, so you still can access it! The code format is `object._<ClassName><hidden attribute name>`, so in our case `example._ExampleClass__color`[5]. However, this is a last resort sort of thing that you should use only if you absolutely must access that attribute or method and, hopefully, know what you doing.

[4] Note that you cannot have write-only property, you must have either getter alone or both.
[5] Try it and see that it works.

You can also come across attributes with a *single* leading underscore in the name, e.g. `_color`. These are not private and are fully visible. However, the leading underscore *hints* that this attribute or method should be *considered* private. So, if you see an attribute like `_color`, you should pretend that you know not of its existence and, therefore, you never read or modify it directly. Of course, this is only an *agreement*, so you can always ignore it and work with that attribute directly[6]. However, this almost certainly will break the code in unexpected and hard-to-trace ways.

16.11 Landing

We should check for landing whenever the bottom edge of the lander is at or below the top edge of the landing pad. A successful landing must satisfy several conditions:

- The lander must be within the limits of the lander pad horizontally.
- The vertical speed must be zero or negative (otherwise, the lander flies up) but below a certain threshold that we will define as a constant `VERTICAL_SPEED_THRESHOLD = 0.05`.
- The absolute horizontal speed must be below a certain threshold, also defined as a constant `HORIZONTAL_SPEED_THRESHOLD = 0.05`.

If *any* of these three conditions are false, the lander has crashed. Either way, the game is over, so you should record the outcome (whether the landing was successful) and set `gameover` to `False`. After the loop, inform the player about the outcome. Draw all game objects plus the message about the outcome (e.g., "You did it!" / "Oh, no!" or something else) and wait for a space key press.

The condition above will be quite long, so fitting it into a single line will make it hard to read. In Python, you can split the line by putting \ at the end of it. So here, a multiline if statement will look as follows:

```
if lander_is_within_horizontal_limits and \
   lander_vertical_speed_is_good and \
   lander_horizontal_speed_is_good:
   ...
else:
  ...
```

Implement landing checks.

[6] On a side note of doing crazy things that you shouldn't: You can replace a class method with your own at run time, this is called "monkey patching"!

16.12 More rounds

Extend the game to have more than one round after the player either landed or crashed. Remember to reset the position of the lander before each new round. You can also add a `reset()` method to the landing pad as well, randomizing it horizontal position. Importantly, *escape* key should quit the game, not just the round, and there should be no "success"/"failure" message in that case. Think how you would implement this.

Add more rounds.

16.13 Limited fuel

Let us add a fuel limit to make things more interesting, so that thrursters would work *only* if there is any fuel left. For this, define a new constant `FULL_TANK` (I've picked it to be `100` but you can have more) and add a new attribute `fuel` to the lander class (remember that you need to explicitly define all attributes in the constructor). The `fuel` level should be set to `FULL_TANK` whenever you reset the lander.

Every use of a thruster should reduce this by 1 and thrusters should work *only* if there is fuel. You need to take care of this in the `update` method. Think about how you would do it for both vertical and horizontal thrusters.

We also need to tell the player how much the fuel is left. I've implemented it as a bar gauge but you can implement it as text stimulus as well. Create the appropriate visual attribute in the constructor and remember to update it every time the level of the fuel changes and to draw it whenever you draw the lander itself. As a nice touch, you can change the color to indicate how much of the fuel is left. I've used *green* for more than 2/3, *yellow* for more than 1/3, and *red* if less than that.

Add fuel and fuel gauge.

16.14 Add to it!

We already have a functioning game but you can add so much more to it: visuals for the thrusters, sounds, background, etc. Experiment at your will!

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.

16.14.0.1 W{- .unlisted}}

- While loop.