Writing games with Python and PsychoPy

Alexander (Sasha) Pastukhov

2021-10-19

Contents

Chapter 1

Introduction

This book will teach you programming. Hopefully, it will do so in a fun way because if there is something more satisfying than playing a video game then it is creating one. Although it is written for the course called *"Python for social and experimental psychology"*, my main aim is not to teach you Python per se. Python is a fantastic tool (more on this later) but it is just one of many programming languages that exist. My ultimate goal is to help you to develop general programming skills, which do not depend on a specific-programming language, and make sure that you form good habits that will make your code clear, easy to read, and easy to maintain. That last part is crucial. 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 and easy-to-read code that others and, even more importantly, you-two-weeks later can understand.

1.1   Prerequisites

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.

1.2   Why games?

The actual purpose of this course is to teach psychology and social studies students how to program *experiments*. That is what the real research is about. However, there is little practical 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 can certainly program an experiment.

9

1.3   Why should a psychologist learn programming?

Why should a psychologist, who is interested in people, learn how to program computers? The most obvious answer is that this is a useful skill. Being able to program gives you freedom to create an experiment that answers your research question, not an experiment that can be implemented given constraints of your software.

More importantly, at least from my point of view, learning how to program changes the way you think in general. People are smart but computers are dumb. When you explain your experiment or travel plans to somebody, you can be fairly vague, make a minor mistake, even skip certain parts. People are smart so they will fill the missing information in with their knowledge, spot and correct a mistake, ask you for more information, and can improvise on their own once they encounter something that you have not covered. Computers are dumb, so you must be precise, you cannot have gray areas, you cannot leave anything to "it will figure it out once it happens" (it won't). My personal experience, corroborated by psychologists who learned programming, is that it makes you realize just how vague and imprecise people can be without realizing it. Programming forces you to be precise and thorough, to plan ahead for any eventuality there might be. And this is a very useful skill by itself as it can be applied to any activity that requires planning be that an experimental design or travel arrangements.

1.4   Why Python?

There are many ways to create an experiment for psychological research. 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 (knowing Python supercharges your PsychoPy experiments). 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. At the same time, you can always opt in, if drag-and-drop is sufficient but faster. Or use a mix of the two approaches. 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. In this seminar, we will concentrate on desktop experiments but you can use it for online experiments (oTree and PsychoPy), scientific programming (NumPy and SciPy), data analysis (pandas), machine learning (keras), website programming (django), computer vision (OpenCV), etc. Thus, Python is one of the most versatile 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 license fees (a very real problem, if you use Matlab).

1.5 Seminar-specific information

This is a material for *Python for social and experimental psychology* seminar as taught by me at the University of Bamberg. Each chapter covers a single game, introducing necessary ideas and is accompanied by exercises that you need to complete and submit. To pass the seminar, you will need to complete all assignments, i.e., write all the games. You do not need to complete or provide correct solutions for *all* the exercises to pass the course and information on how the points for exercises will be converted to an actual grade (if you need one) or "pass" will be available during the seminar.

The material is structured, so that each chapter or chapter section correspond to a single meeting. However, we are all different, so work at your own pace, read the material and submit assignments independently. I will provide detailed feedback for each assignment and you will have an opportunity to address issues and resubmit again with no loss of points. Note that my feedback will cover not only the actual problems with the code but the way you implemented the solution and how clean and well-documented your code is. Remember, our task is not just to learn how to program a working game but how to write a nice clear easy-to-read-and-maintain code[1].

Very important: Do not hesitate to ask questions. If I feel that you missed the information in the material, I will point you to the exact location. If you are confused, I'll gently prod you with questions so that you will solve your own problem. If you need more information, I'll supply it. If you simply want to know more, ask and I'll explain why things are the way they are or suggest what to read. If I feel that you should be able to solve the issue without my help, I'll tell you so (although, I would still probably ask a few hinting questions).

---

[1]Good habits! Form good habits! Thank you for reading this subliminal message.

1.6 About the material

This material is **free to use** and is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives V4.0 International License.

Chapter 2

Software

For this book and seminar, we will need to install

- PsychoPy that comes bundled with Python.
- IDE of your choice. My instructions will be for Visual Studio Code, which has a very good Python support.
- Jupyter Notebook for trying out small snippets of code.

I will not give detailed instructions on how to install the necessary software but rather point you to official manuals. This makes this text more future-proof as specific details might easily change[1].

2.1   PsychoPy

Download and install Standalone PsychoPy version. Use whatever the latest (and greatest) PsychoPy version is suggested to you (PsychoPy 2021.2.3 using Python3.6 as of time of writing) and follow instructions.

Note that you can also install PsychoPy as a anaconda package or install an official Python distribution and add PsychoPy via pip. However, I find the standalone easier to use as it has all necessary additional libraries. Plus, it has additional tools for GUI-based experiment programming and integration with Pavlovia.org.

---

[1]If you are part of the seminar, ask me whenever you have problems or are unsure about how to proceeed

2.2   VS Code

Visual Studio Code is a free lightweight open-source editor with strong support for Python. Download the installer for your platform and follow the instructions.

Next, follow Getting Started with Python in VS Code tutorial. **Skip** the *Install a Python interpreter* section, as you already have Python installation bundled with PsychoPy. This is the interpreter that you should use in the *Select a Python interpreter* section. In my case the path is `C:\Program Files\PsychoPy3\python.exe`.

Install and enable a linter, software that highlights syntactical and stylistic problems in your Python source code. Follow the manual at VS Code website.

2.3   Jupyter Notebooks

Jupyter Notebooks offer a very convenient way to mix text, figure and code in a single document. They also make it easy to play with various small snippets in parallel without running scripts. We will rely on them for our first chapter and for an occasional exercises or code testing later on. There are two way you can use them: 1) in VS Code using Jupyter extension, 2) in your browser using classical interface.

2.3.1   Jupyter Notebooks in VS Code

Follow the manual on how to install Jupyter package and use notebooks in VS Code.

2.3.2   Jupyter Notebooks in Anaconda

The simplest way to use Jupyter Notebooks along with a lot of other useful data science tools is via Anaconda toolkit. However, note that this will introduce a *second* Python distribution to your system. This, in turn, could lead to some confusion when working with scripts in VS Code if you accidentally have Anaconda interpreter active instead of the PsychoPy one. Do not panic, follow Select a Python interpreter instructions and make sure that you have PsychoPy interpreter as the active one.

Otherwise, download and install Anaconda. The website has an excellent Getting started section.

2.4 Keeping things tidy

Before we start, I suggest that you create a folder called *games-with-python* (or something along these lines). If you opted to use Jupyter Notebooks via Anaconda, you should create it in your user folder because this is where Anaconda would expects to find them. Then, create a new subfolder for each chapter / game. For the seminar, you would need to zip and upload a folder with all the files.

Chapter 3

Programming tips and tricks

Below are some tips about writing and reading the code. Some may sound cryptic when you read them for the first time (they will become clear once we cover the necessary material). Some will feel like an overkill for simple projects that we will be implementing. I suggest that you read this section casually the very first time but return to it frequently once we start to program in earnest. 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. The seat belt does nothing useful on most (hopefully, all) days but you wear it because it might suddenly and very urgently 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, a pilot experiment, etc. But, as they say in Russia "There is nothing more permanent than a temporary solution". More often than not, you will 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 you did a few months ago 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!

---

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

3.1   Writing the code

3.1.1   Use a linter

Linter is a program that analyses your code *style* and highlights any issues
it finds: spaces where should be none, no spaces where should be some, wrong
names, overly long lines, etc. These do not affect how the code runs but following
linter's advice results in a consistent standard if boring-looking[2] Python code.
Try to address all the problems that the linter raised. However, use your better
judgment because sometimes lines that are longer than linter would prefer are
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.

3.1.2   Document your code

Every time you create a new file: document it and update the documentation
whenever you add/change/delete new functions or classes. Every time your
create a new function: document it. New class: document it. New constant:
unless it is super clear from the name alone, document it. You will learn a
NumPy way of doing this in the book.

I cannot stress how important documenting your code is. VS Code (an editor
that we will use) is smart enough to parse NumPy docstring, so it will show
this help to you whenever you use your own functions (helps you to help you!).
More importantly, writing documentation forces you to think and formulate (in
human language!) what the function or class is doing, what type the arguments /
attributes / methods are, what is the range of valid values, what are the defaults,
what should a function return, etc. More often than not, you will realize that
you have overlooked some important detail that may not be apparent from the
code itself.

3.1.3   Add some air

Separate chunks of code with some empty lines. Think paragraphs in the normal
text. You wouldn't want 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 will be a point when we draw
all stimuli and redraw the window. 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 to understand what this chunk is about and whether it is relevant for

---

[2]"Boring is Good!", see "The Hitman's Bodyguard" movie.

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 and no code that does something else!

3.1.4   Write your code one teeny-tiny step a time

Your motto should be "slow but steady". This is the way I will guide you through the games. Always start with a something extremely 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[3] 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 it work. It is much easier to solve simple problems one at a time than a lot of them simultaneously.

3.1.5   There is nothing wrong with StackOverflow

Yes, you can always try to find a solution to your problem on StackOverflow[4]. I do it all the time! However, you should use the provided solution *only if you understand it*! Do not copy-paste the code that *seems* to solve a problem like yours. If you do that and you are lucky, it might work. Or, again if you are lucky, it won't work in an obvious manner. But if you are not so lucky, it will (sometimes) work incorrectly in a subtle way. And, since you did not really know what the code was doing when you pasted it, you will be even more confused. So use StackOverflow as a source of knowledge, not as a source of copy-pastable code!

Reading the code {#reading-tips} Reading code is easy because computers are dumb and you are smart. This means that instructions you give the computer must necessarily be very simple and, therefore, are very easy to understand for a human. Unfortunately, reading code is also hard because computers are dumb and you are smart. Your are so smart that you don't even need to read the entire code to understand what it is doing, you just read the key bits and fill in the gaps. Unfortunately, this means that you will tend to read over mistakes. This is not unique to programming, if you ever proofraed a text, you now how hard it is to find tipos. Your brain corrects them on the fly using the context and you read the word as it should be, not as it is actually written[5].

---

[3]Seriously, 100%! If you have even a shadow of a doubt, check again. That shadow will grow and make you progressively uncertain about your code.

[4]However, if you are doing the seminar, ask me first!

[5]Tip: Read your text one sentence at a time starting from the back or read one random sentence at a time. This breaks the flow of the text and helps you concentrate on words rather

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 and obvious in retrospect[6]. Do not despair! It is not you, but just a consequence of how wonderfully your brain is wired for pattern-recognition. Below are several suggestions that could help you to make reading code more robust.

3.1.6   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 and 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.

3.1.7   Pretend that you've never seen this code in your life

Assume that you have no idea what the code is doing. As I wrote, 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. Turning your expectations off is hard but is immensely helpful.

3.1.8   Do not search only under the street lamp

Whenever you are using some new code or need to implement something that feels complicated and your code does not work as it should, you will tend to assume that a problem is with the new fancy code. Simply because it is new, fancy, and complicated. But, in my experience, the error will typically hide in plain sight in the simpler "trivial" code nearby which you never properly look at, because it is simple and trivial. Check everything, not just the places where you would expect to have made a mistake.

3.1.9   Use the debugger

In the book, you will learn how to pause an execution of your game, so you can investigate its state. Use this knowledge! Put breakpoints and execute the code step-by-step. Check values of variables using "Watch" tab. Use debug console to check whether functions return results that they should. For complex conditions or mathematical formulas, split them into small bits, copy and execute these

---

than on the meaning and the story.

[6]Hindsight is always 20/20!

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.

3.2 Zen of Python

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

Chapter 4

Python basics

Hopefully, you already created a special folder for this book. Download the exercise notebook (Alt+Click should download rather than open it), put it in a chapter's folder, and open it (see relevant instructions. You will need to switch between explanations here and the exercises in the notebook, so keep them both open.

4.1 Chapter concepts

- Variables.
- Constants.
- Basic value types.
- Printing things out.
- Putting values into strings.

4.2 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 that a game character has. We begin with a number typical for a human being.

In Python, you would write

```python
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`.

```python
# 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 (but what if our character has only one leg?)
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.

```python
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 a new value overwrites the old one. Check yourself, what will be final value of the variable in the code below?

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

Do exercise #1.

Note that a variable ("a box with a name tag") exists only after you assign something to it. So, you the following code will generate a `NameError`, a Python's way to tell that you it never heard of variable `number_of_hands`.

```python
number_of_legs = 2
number_of_gloves = number_of_hands / 2
```

However, you can create a variable the does not hold any *specific* value by assigning `None` to it. `None` was added to the language specifically to mean *no value* or *nothing*.

```python
number_of_hands = None # variable exists now, but holds no particular value.
```

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.

4.3   Assignments are not equations!

Very important: although assignments *look* like mathematical equations, they are **not equations!** They follow a **very important** rule that you must keep in mind when understanding assignments: the right side of an expression is evaluated *first* until the final value is computed, then and only then that 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? First, it takes *current* values of all variables (2 for x and 5 for y) and puts 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.

4.4   Constants

Although the real power of variables is that you can change their value, you should use them even if the value remains constant throughout the program. 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 agreements (such as a `snake_case` convention above). We will encounter more of such agreements later, for example, 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 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 properly named 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 permanently *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[1] and definitely long search-and-replace throughout the entire code.

Do exercise #4.

4.5 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`.

---

[1]not really

When using a variable it is important that you know what type of value it stores and this is mostly on you. In some cases, Python will raise an error, if you try doing a computation using incompatible value types. In other 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?

```python
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?

```python
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?

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

secret_message = ''
bool(secret_message)

bool('False')
```

Do exercise #7.

Converting to integer or float numbers using, respectively, int(<value>) and float(<value>) 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 (it does not do proper 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 an 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?

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

Do exercise #8.

4.6 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.

```python
print(5)
#> 5
```

or

```python
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.

```python
number_of_pancakes = 10
print(number_of_pancakes)
#> 10
```

or

```python
breakfast = "pancakes"
print(breakfast)
#> pancakes
```

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

```python
breakfast = "pancakes"
number_of_items = 10
print(breakfast, number_of_items)
#> pancakes 10
```

What will be printed by the code below?

```python
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:

```python
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.

4.7   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 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)`. You will learn about tuples later. For now, assume

that it is just a comma-separated list of values enclosed in round brackets: `(1, 2, 3)`.

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:

```python
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 in the right order. Before running it, can you figure out which call will actually work (and what will be the output) and which will produce an error?

```python
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.

As noted above, 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 in the next chapter.

Chapter 5

Guess the Number: a single round edition

The previous chapter covered Python basics, so now you are ready to start developing you first game! We will build it step by step as there will be a lot to learn about input, libraries, conditional statements, and indentation.

Before you start, create a new folder (inside your course folder) called, for example, "guess-the-number", download exercise notebook, copy it in the newly created folder, and open it in Jupyter Notebook. As in the chapter before, it will contain exercises on reading and understanding the code.

However, we will be using VS Code to program scripts with the actual game. You will need to create a separate file for each code practice[1] (e.g., *code01.py*[2], *code02.py*, etc.) This is not the most efficient implementation of a version control and it will certainly clutter the folder. But it would allow me to see your solutions for every step, which will make it easier for me to give feedback. For submitting the assignment, just zip the folder and submit the zip-file.

5.1   Chapter concepts

- Documenting code.
- Debuggin code.
- Getting input from a user.
- Using comparison in conditional statements.
- Using indentation to group statements together.

---

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

[2]I recommend using `01` instead of `1`, as it will ensure consistent file sorting in your file manager

- Using Python libraries.
- Generating random numbers.

5.2   The Game

We will program a game in which one participant (computer) picks a number within a certain range (say, between 1 and 10) and the other participant (human player) is trying to guess it. After every guess of the human, the 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 guesses the number or (in the later version of the game) runs out of attempts.

Our first version will allow just one attempt and the overall game algorithm will look like this:

1. computer generates a random number between 1 and 10
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!"

5.3   Let's pick a number

Let us start by implementing just the first two steps of the program. First, create a variable holding a number that computer "picked". We will name it `number_picked` (you can use some other meaningful name as well but it might be easier if we all stick to the same name). To make things a simpler at the beginning, we will hard-code an arbitrary number between 1 and 10 (pick the one you like). Then, let us print it out, so that we know the number ourselves [3]. Use string formatting to make things user-friendly, e.g., print out something like "The number I've picked is …". You should be able to do this using the knowledge from the previous chapter. Your code should be a two-liner:

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

Try out this two-liner in a Jupyter Notebook (create an empty notebook just for that). Once you are happy with it, copy-paste the code into `code01.py` and read on to learn how to run it.

Put your code into `code01.py`.

---

[3]Of course, we know it because we hard-coded it. But that won't be the case when computer will generate it randomly, so let us plan for the future

5.4 Documenting your code

Now that you have your first file with Python program, you should document it. Documenting a two-lines long and simple program may feel silly but it should be an automatic thing. Later on you will find yourself using several lines of comments to document a single line function. Again, it is not about the code that works, it is about the code you can understand. In a sense, it is better to have a clean well-documented program that currently does not work correctly than an undocumented spaghetti code that does. You can fix and update the former, but maintaining or updating the latter…

In Python, you have two ways to write comments multiline and single line

```python
'''
A
multiline
comment
'''

#  A single line comment.
```

Use multiline comments to write documentation for individual files, functions, classes, methods, etc. You will learn how to format this documentation Numpy docstring style, once you learn about functions. In our case, you should start your `code01.py` file with a multiline comment that briefly describes what program it contains. Minimally, you should write that this is a *Guess a Number* game. It is probably a good idea to sketch out what the game is about.

Use single line comments to explain what happens in a particular code block. You do not need to comment every line and you should not explain the code in human language. A comment should be about *what* is going on not *how*. At the moment, we have too little code to need this.

Document `code01.py` file.

5.5 Running and debugging your game in VS Code

Now that we have a two-lines-long program, we can run it and already start learning how to debug it. Yes, our current program is probably to simple to require actual debugging but it is also simple enough to make understanding debugging easier as well. And debugging is a crucial skill that turns a running program from a black box into something transparent and easy[4] to understand.

---

[4]Or, at least, easier.

Below I will describe how to debug in VS Code but you might want to consult an official manual in case things have changed in the meantime.

There are two ways to run a Python program in VS Code. First, you can use the *"Run Python File in Terminal"* play button on the right. This runs your program *without* a debugger, so you should use it only for the actual runs of the finalized code. Still, you can try running it and see whether it prints out what it should.
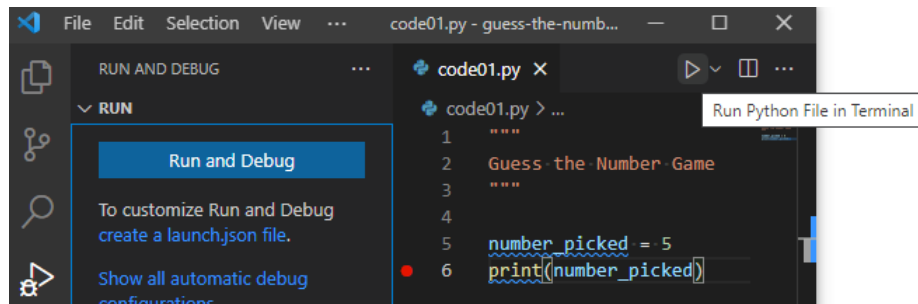
Figure 5.1: Running code without a debugger.

The alternative is the debugging tab, the one with a little bug on top of the run button. If you select it, it will show *"Run and Debug"*. Click on the button and it will offer various choices for different kinds of Python projects and frameworks. For our intents and purposes, we will only need *"Python File: Debug the currently active Python file"*. Click on that and it will execute your code (should run exactly the same way as with the other button).
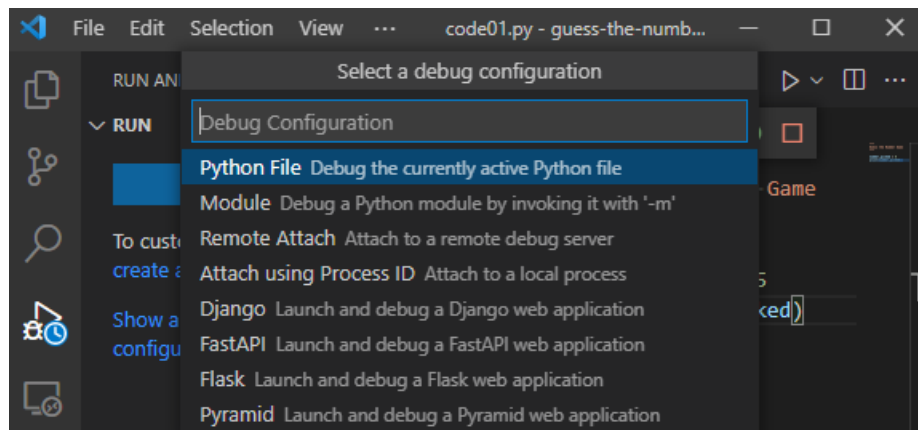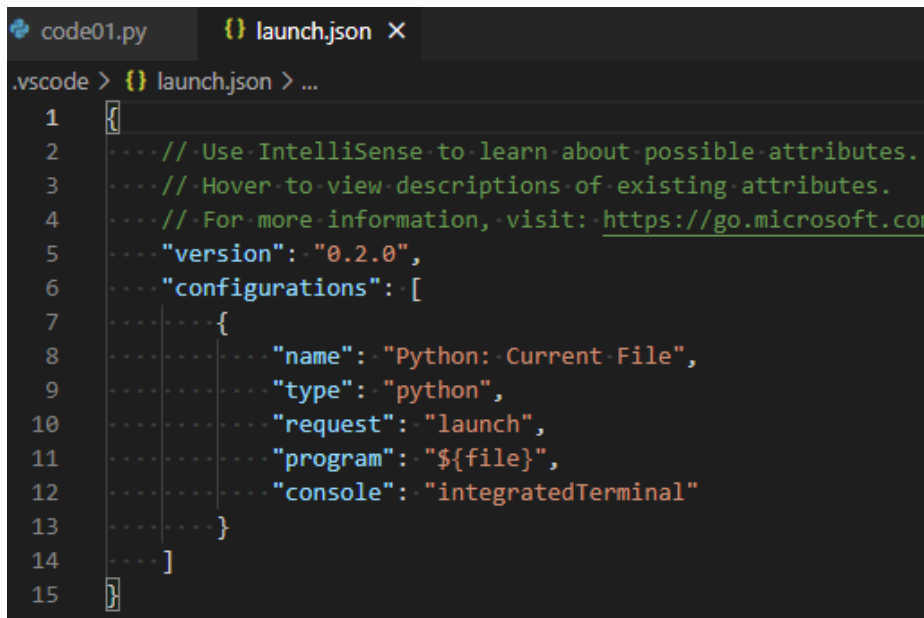
Figure 5.2: Selecting debugging configuration.

You probably do not want to click through Debugging tab → Run and Debug button → Pick configuration all the time. A better way is to configure it once

and for all and then use **F5** key to run your code. First, click on *"create a launch.json file"* and select the *"Python File: Debug the currently active Python file"* again. You will see a new *launch.json* file appearing in the editor that should look like this:

Figure 5.3: Debug configuration in launch.json file.

That's it! VS Code created a configuration for you. Now you can close *launch.json* file and run your program with a simple press of **F5** button. Try it! Again, it should work just like before, so why did we go through all this trouble? Because debugger will *pause* an execution of your code whenever it encounters a problem, giving you a chance to examine variables, run snippets of code, etc. In contrast, running python file in terminal (the first option) will only print an error message and exit the program. Moreover, you can use *breakpoints* to pause the program at any line, which gives you an opportunity to examine your code at any location that you need.

You enable breakpoints by clicking to the left of the line number that you are interested in. Here, I've clicked on line 6 and you can see a red dot that indicates an active breakpoint.

If I now run the code via **F5**, the program will stop at that line *before* executing it.

This gives me a chance to see what value my `number_picked` variable has. It is already listed in local variables (top left). But I also added it to list of watched variables (*Watch*, lift middle) and I also looked at its value in the

Figure 5.4: Active breakpoint.

Debug Console (bottom tab) that allows me to execute *any* Python code while my program is paused. Do the same and explore these different ways yourself. For example, see how you can compute `number_picked + 1` or `number_picked * number_picked` in the *Watch* tab and in *Debug Console.*

Once you are done examining the current state of the program, you have six buttons at the top to decide what to do next (hover over them to see hints). They are, from left to right

- Continue (**F5**): resume the program.
- Step Over (**F10**): executes the code without going into functions (this and two following options will become clearer once you learn about writing functions).
- Step Into the code (**F11**)
- Step Out of the code (**Shift+F11**).
- Restart the program (**Ctrl+Shift+F5**).
- Stop the program (**Shift+F5**).

To better understand how this works, stop the program (**Shift+F5**) and put an additional breakpoint on the first line of your code (line #5 in my program, the other lines are comments or are empty). Run the program again via **F5** and it will pause at that very first line. Can you figure out the value of variable `number_picked` now?

The answer is "no" because that variable does not exist yet. Remember, the program pauses *before* executing the line. Use **F10** to step through the code line by line to see how the variable appears and the information gets printed out.

Figure 5.5: Program paused at the breakpoint.

This debugging exercise may not have been necessary to solve problems with your current code but it showed how to do that in the future. Do not hesitate to put a breakpoint to pause the program and check that reality (i.e., actual values of variables) matches your expectations. Use the stepping through the code to slow things down to watch and ponder.

5.6   Asking a player for a guess

It takes two to play the *Guess the Number* game. The computer did its part by picking a number, now we need to ask a player to enter their guess. For this, we will use input([prompt]) *function*.

A function is an isolated code that accepts (optional) *inputs*, performs an *action*, and, optionally, returns a value (*output*). This allows both to split the code into smaller chunks that are easier to maintain and to reuse the same code. You already used print() function to print things out, and str(), bool(), int() and float() functions to convert values. For print() *input* is an arbitrary number of values (even none, try this in Jupiter Notebook!), its *action* is to print things out, but it returns nothing (no *output*). float() function takes (somewhat surprisingly) zero or one value as *input* (try giving it none or more than one in Jupiter Notebook as see the difference), attempts to convert given value to float (throwing an error, if it cannot do it), and returns a float value as an *output*.

Similar *input* → *action* → *output* scheme holds for the input([prompt]). It takes optional `prompt` string as input. Then it prints the `prompt` message and waits for a user to enter a *string* until they press **Enter**. It then returns this *string* value. The latter bit about *string* is important, because in our game we need a player to enter an *integer* not a string. For a moment, let us assume that the input is always a valid integer number, so type only valid integers when testing the program! This way we can convert it to an integer without extra checks (will add them in the future) and assign this value to a new variable called `guess`. Thus, you need to add a single line assignment statement with `guess` variable on the left side and call to input() function on the right side (think of a nice prompt message) wrapped by (inside of) the type-conversion to integer via int(). Test this code but, again, only enter valid integers, so that the conversion works without an error.

Update your `code01.py`.

5.7   Conditional *if* statement

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

```python
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:

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

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

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

5.8 Conditions and comparisons

Condition is any expression that can be evaluated to see whether it is `True` or `False`. A straightforward example of such expression are comparisons expressed in human language as: *Is today Thursday? Is the answer (equal to) 42? Is it raining and I have an umbrella?*. We will concentrate on comparisons like that for a moment but later you will see that in Python *any* expression is either `True` or `False`, even when it does not look like a comparison[5].

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 as `=>` and `=<` will produce an error).

---

[5]This is because you can convert any value to a logical one via bool() function that you learned about the last time and so any value is (converted to) either `True` or `False`.

Go to exercise #1 to solve some comparisons.

Note that Python also has an is operator that may *look* identical to `==` (e.g., `x == 2` looks equivalent to `x is 2`). Moreover, is *some* cases it also works the same way. However, there is a subtle difference: `==` checks whether *values* are identical, whereas `is` checks whether *objects* (that "hold" values) are identical. You need to understand classes and objects before you can appreciate this difference, so for now just keep in mind that you should only use `==` (I will explicitly mentioned when `is` is needed).

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 #2 to explore this inversion yourself.

You can also combine several comparisons using `and` and/or[6] `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 linked via `and`: **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 #3 and #4.

Subtle but important point: conditions are evaluated from left to right until the whole expression is resolved one way or another. 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 `first` expression already 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")` alone would raise a `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

---
[6] pun intended

```
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 #5.

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 #6.

Final thought on comparisons: Do not hesitate to test them in Jupyter Notebook using various combinations of values or pause the program at the condition via a breakpoint and evaluate a comparison in *Watch* or *Debug Console*.

5.9 Grouping statements via indentation

Let us go back to a conditional if-statement. Take a look at following code example (and note an `:` at the end of the `if some_condition_is_true:`), 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** that are automatically added whenever you press **Tab** in VS Code and removed whenever you press **Shift+Tab**. The indentation puts statement #1 *inside* the

if-statement. Thus, indentation shows whether statements belong to the same group and must be executed one after another (same indentation level for `if` and `statement #2`) or are inside conditional statement, loop, function, class, etc. (`statement #1`). For more complex code that will have, for example, an if-statement inside an if-statement inside a loop, you will express this by adding more levels of indentation. E.g.

```python
# 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! A wrong indentation level is unfortunately a very common mistake.

Do exercise #7.

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 #8.

5.10   Checking the answer

Now you have all necessary tools to finish the first version of our game. Add a conditional statements to your `code01.py`, 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!"`

Update `code01.py`

Test that your code works. Again, use breakpoints if you need to better understand the control flow and check whether comparisons work the way you expect.

5.11 Using libraries

Our game is "feature-complete": computer picks a number, player makes a guess, computer responds appropriately. However, currently we are playing for both sides. Lets make computer pick a random number itself. For this we need to use randint(a, b). It is a part of any Python distribution, so you would have it even if you would install a vanilla Python distribution instead of using one from PsychoPy. However, you cannot use it straightaway like you did with `print()` or `input()`. Type `randint(1, 3)` in your Jupyter Notebook and observe *NameError: name 'randint' is not defined.*

The reason for this is that Python has an awful lot of functions and loading them all at the same time would clog the memory with things you never intended to use. Instead, they are packaged into *libraries*, so that you can import only functions (or libraries) that you actually need for your program. You *import* them via an `import` statement that should go to the top of your file (but below the comment about the file content). There are several ways you can import libraries. First, you can import an *entire* library (such as *random* library that has function `randint()` that we need) and then use its functions as `<library>.<function>`. For `randint` this would be

```python
import random

computer_pick = random.randint(1, 5)
```

I would suggest this as a preferred way of using libraries as it forces you to explicitly mention the name of the library when calling a function, i.e. `random.randint()` instead of just `randint()`. This may not look important with just one imported library but even in a modestly-sized project you will import many libraries, so figuring out which library the function belongs to will be tricky. Even more importantly, different libraries may have functions with *the same name*. In this case, the function you are using will come from the *last* library you imported. But you may not realize this and this is a sort of mistake that is really hard to track down. Thus, unless you have a very good reason to do otherwise, always import the entire library and use `library.` notation!

Another and less explicit option is to import only *some* functions and use them *without* the `library.` prefix. You can import more than one function by listing them all

```python
from random import randint, randrange

computer_pick = randint(1, 5)
```

You can also *rename* a library or a function while importing it via `as`. In short, you should not do this as using a different name for a library or a function would make it harder for others (and even for future-you) to understand your code. However, there are some "standard" renaming patterns that are used universally and which you are likely to encounter.

```python
# this is a standard way to import these two libraries
import numpy as np
import pandas as pd

np.abs(-1)

# you can rename individual functions as well, if you really have to (but, please, don
from random import randint as random_integer

computer_pick = random_integer(1, 5)
```

Finally, there is a **very bad way** to import functions from a library: `from random import *`. The asterisk means that you want to import *all* functions from the library and you want to call them by their names without `random.` prefix. Never, never, never[7] do this! This fills your environment with functions that you may not be aware of, potentially overwriting some other functions, creating conflicts, etc. Never ever! I am showing you this only because you will, at some point, see a code that uses this approach and you might think that this is a good idea. It is a terrible idea! Import the library, not the functions, so you can explicitly show which library you are relying on when calling a function. Always remember the Zen of Python: "Explicit is better than implicit."

5.12   Picking a number randomly

Now that you know how to import a library, we can use randint() function. For this, save a copy of your original code into `code02.py`. Import the library and use randint() to generate a random number between 1 and 10. Read the documentation on randint() to understand how to use it. Reading manuals is a necessary part of programming, so this is a good point to start practicing.

Once you implemented this in `code02.py`, run it several times to check that computer does pick different random values. Again, use breakpoints if you want to double-check what is going on.

---

[7]Did I already say never? never!

Put your code into `code02.py`.

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

5.13 One-armed bandit (a single round edition)

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

1. Import random library, so you could use `randint` function
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!"`.
   - Do not print anyting, if all numbers are different.

Do not forget to document the new `code03.py` file and feel three to use breakpoints to debug it.

Put your code into `code03.py`.

5.14 Submitting for the seminar

For the seminar, submit a zipped folder with exercise notebook and all three programs.

Chapter 6

Guess the Number: a multi round edition

In previous chapter, you programmed a single-attempt-only "Guess the Number" game. Now, we will expand to allow multiple attempts and will add other bells-and-whistles to make it more fun. Create a new subfolder and download the exercise notebook before we start!

6.1 Chapter concepts

- Repeating code using while loop.
- Making in emergency exit from a loop.

6.2 While loop

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

```python
# 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. Also, the same indentations rules determine which code is inside the loop and which outside.

Do exercise #1.

Let us use *while* loop to allow the player to keep guessing until they finally get 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

```
# import random library so you can use randint function

# generated a random number and store in number_picked variable
# get player input, convert it to an integer, and store in guess variable

# 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
    # get player input, convert it to an integer, and store in guess variable

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

Put your code into `code01.py`.

Do not forget to document the file and use breakpoints and step overs to explore the program flow.

6.3   Counting attempts

Now let us add a variable that will count a total number of attempts by the player. 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 enters a guess. After the loop, expand the `"Spot on!"` message by adding information about the number of attempts. Use string formatting to make things look nice, e.g., `"Spot on, and you needed just 5 attempts!"`. Check that the number of attempts your required *matches* the number of attempts reported by the program!

Put your code into `code02.py`.

6.4 Breaking (and exiting)

Code inside the *while* loop is executed repeatedly while the condition is `True` and, importantly, all of code the 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 has a break statement that causes the program to exit the loop immediately without executing the rest of the code inside the loop, so that the program continues with the code *after* the loop.

```python
# 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 #2 to build your intuition.

6.5 Limiting number of attempts via break

Let's put the player under some pressure! Decide on maximal number of attempts you allow and stores it as 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 the current attempt number is greater than `MAX_ATTEMPTS`. Think about when (within the code inside the loop) you should check this.

Put your code into `code03.py`.

6.6 Correct end-of-game message

Let us update the final message. Currently it says "Spot on…" because we assumed that program exited the loop only if the player gave a correct answer. With limited attempts that is not necessarily the case. Now there are two reasons why it exited the while loop:

1. The player answered correctly

2. The player ran out of attempts.

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

Put your code into `code04.py`.

6.7   Limiting number of attempts without a break

Although it was my idea to add the `break` statement, you should use it sparingly. Without `break` there is a *single* place in the code that you need to check to understand when the program will exit the loop: the condition. However, if you add a `break`, you now have *two* places that need to be examined. And every additional `break` keeps adding to that. This does not mean that you should avoid them at all costs! You *should* use them, if this makes the code easier to understand. But always check if a modified condition could also do the trick.

Let us try exactly that. Modify your code to work *without* the `break` statement. You need a more complicated condition for your while loop. so that it repeats while player's guess is incorrect and the number of attempts is still less than the maximally allowed. Test that your code works both when you win and when you lose.

Put your code into `code05.py`.

6.8   Show remaining attempts

It is all about a user interface! Modify the `input` prompt message to include a number of *remaining* attempts. E.g. `"Please enter the guess, you have X attempts remaining"`.

Put your code into `code06.py`.

6.9   Repeating the game

Let us give an option for the player to play again. This means putting *all* the current code inside of another `while` loop (this is called *nested loops*) that is repeated for as long as the player wants to keep playing. The code should look following:

```
# import random library so you can use randint function

# define MAX_ATTEMPTS

# define a variable called "want_to_play" and set to True
# while the player still wants to play

  # your current 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" or "y"

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

Pay extra attention to indentations to group the code properly!

Put your code into `code07.py`.

6.10   You do not need a comparison, if you already have the value

In your updated code, you have `want_to_play` variable that is either `True` or `False`. It is used in the loop that repeats while its value is `True`. Sometimes, people write `want_to_play == True` to express that. While it is technically correct and will certainly work correctly, it is also redundant. Since `want_to_play` can only be `True` or `False` this comparison turns into `True == True` (which is of course `True`) or `False == True` (which is `False`). So comparing either value to `True` produces exactly the same value. Thus, you can just write `while want_to_play:` and use the logical value directly.

6.11   Best score

A "proper" game typically keeps the track of players' performance. Let us record a 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. You should update it after each game round. Add the information about "Best so far" into the round-over message.

Put your code into `code08.py`.

6.12   Counting game rounds

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 `code09.py`.

6.13   Wrap up

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

Chapter 7

Guess the Number: AI takes a turn

Let us program Guess the Number game again[1] but *reverse* the roles. Now *you* will pick a number and the computer will guess. Think about the algorithm that a computer could use for this before reading the next paragraph[2].

The optimal way to do this is to use the middle of the interval for a guess. This way you rule out *half* the numbers that are either greater or smaller than your guess (or you guess the number correctly, of course). So, if you know that the number is between 1 and 10, you should split things in the middle, that is picking 5 or 6, as you cannot pick 5.5 (we assume that you can use only integers). If your opponent tells that their number is greater than your pick, you know that it must be somewhere between your guess and the original upper limit, e.g., between 5 and 10. Conversely, if the opponent responds "lower", the number is the lower limit and you guess, e.g., between 1 and 5. On your next attempt, you pick split the interval again until you either guess the number correctly or end up with an interval that contains just one number. Then you do not need to guess anymore.

To implement this program, you will need to learn about functions, how to document them like a pro, and how to use your own libraries.

7.1 Chapter concepts.

- Writing you own functions.
- Understanding variable scopes.

---

[1]This is the last time, I promise!
[2]You should imagine Dora the Explorer staring at you while you think.

- Adopting standard ways to document your code.
- Using your own libraries.

7.2   Player's response

Let us warm up by writing a code that will allow a player to respond to computer's guess. Recall that there are just three options: your number is greater, smaller, or equal to a computer's guess. I would suggest using, respectively, `>`, `<`, and `=` symbols to communicate this. You need to write the code that will prompt a player for their response until they enter one of these symbols. I.e., the prompt should be repeatedly repeated if they enter anything else. Thus, you definitely need to use the input([prompt]) and a while loop. Think of a useful and informative prompt message for this. Test that it works. Using breakpoints might be very useful here.

Put your code into `code01.py`.

7.3   Functions

You already now how to use function, now it is turn for you to learn more about why you should care. The 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.

7.3.1   Function performs a single computation

I already told you that reading code is easy because every action has to be spelled-out for computers in a 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 two loops with conditional statements nested inside. Add a few more of those and you have so many branches to trace, you never be quite sure what will happen. This is because our cognition and working memory, which you use to trace all branches, are limited to just about four items[3].

Thus, a function should perform *one* computation / action that is conceptually clear and those purpose should be understood directly from its name or, at most, from a single sentence that describes it[4]. The name of a function should

---

[3]The official magic number is 7±2 but reading the original paper tells you that this is more like four for most of us

[4]This is similar to scientific writing, where a single paragraph conveys a single idea. 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.

typically be a *verb* because function is about doing an action. 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 or how the its behavior can be modified. Still, 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 this so much because conceptually simple single-job functions are a foundation of a clear robust reusable code. And future-you will be very grateful that it has to work with easy-to-understand isolated reliable code you wrote.

7.3.2 Function isolates code from the rest of the program

Isolation means that your code runs 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[5] or to variables defined inside of other functions. Conversely, neither global script nor other functions have access to variables and values that you use 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. The 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).

7.3.3 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 produce the correct results? Do the fail clearly and raise a correct 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

---

[5]This is not strictly true but that will concern us only once we get to so-called "mutable" objects like lists or dictionaries.

of automatic unit testing[6] that ensures reliable code for absolute majority of programs and apps that you use[7].

7.3.4 Function makes code reusable

Sometimes, this is given as a 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 approach is a terrible idea as it means that you have to maintain the same code at many places and you might not be even sure in just how many. This is a problem even if a code is extremely simple. Here, we define a *standard* way to compute an initial by taking the first symbol from a string (you will learn about indexing and slicing in details later). The code is as simple as it gets.

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

Imagine that you decided to change it and use first *two* symbols. Again, the computation is not 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 (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 at just *one* location. Here is the original code implemented via a function.

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

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

    Returns
    ----------
    str : single symbol
    """
```

---

[6]it is normal to have more code devoted to testing than to the actual program

[7]You still need tests for the integrated system but testing individual functions is a clear prerequisite.

```python
    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*

```python
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 chunk to worry about and you can be sure that the same computation is performed whenever you call the function (and that these are not several copies of the code that might or might not be identical).

Note that I put reusable code as the last and the least reason to use functions. This is because the other three reasons are far more important. Having a conceptually clear isolated and testable code is advantageous even if you call this function only once. It still makes code easier to understand and to test and

helps you to reduce its complexity by replacing chunks of 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 *compute_initial* tells you what is happening, as it disambiguates the purpose. I suspect that future-you is very pro-disambiguation and anti-confusion.

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

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

7.4   Functions in Python

7.4.1   Defining a function in Python

A function in Python looks like this (note the indentation and : at the end of the first line)

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

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

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

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

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

```python
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()

# And again but via another_function call
another_function()
```

Do exercise #1.

You must also keep in mind that redefining a function (or defining a technically 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 easy to spot, in a large code that spans multiple files and uses various libraries, solving the same problem may not be so straightforward!

7.4.2 Function arguments

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

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

However, you may need to pass information to the function via arguments in order to influence 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:

```python
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!):

```python
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[8].

Do exercise #3.

When you call a function, 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

```python
# hardcoded
print_age(1976, 2020)

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

---

[8]Again, this not strictly true but you will have to wait until you learn about named parameters and default values

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


print_age(1976, get_current_year())
```

7.4.3 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 returns the age instead of printing it (we can always print it ourselves).

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

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

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

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

Do exercise #4.

7.4.4 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 variables defined *inside a function*, exists and are 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[9].

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

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

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


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

Do exercise #5.

7.5   Player's response as a function

Let us put all that theory about functions into practice. Use the code that you created to acquire player's response and turn it into function. I suggest that you call it `input_response` (or something along these lines). Test that the code works by calling this function for the main script.

Put your code into `code02.py`.

---

[9]Again, almost, as *mutable* objects like lists are more complicated, more on that later

7.6 Debugging a function

Now that we have your first function, you can make sense of three step over/in/out buttons that the debugger offers you. Copy-paste the following code in a separate file (call it `test01.py`, for example).

```python
def f1(x, y):
  return x / y

def f2(x, y):
  x = x + 5
  y = y * 2
  return f1(x, y)

z = f2(4, 2)
print(z)
```

First, put a break point on the line in the main script that calls function `f2()`. Run the debugger via **F5** and the program will pause at that line. If you now press **F10** (step over), the program will go to the next line `print(z)`. However, if you are to press **F11** (step into) instead, the program will *step into* the function and go to `x = x + 5` line. When inside the function, you have the same two choices we just looked at but also, you can press **Shift+F11** to step out of the function. Here, the program will run all the code until you reach the next line *outside* of the function (you should end up at `print(z)` again). Experiment with putting breakpoints at various lines and stepping over/in/out to get a hang of these useful debugging tools.

Now, put the breakpoint inside of `f1()` function and run the code via **F5**. Take a look at the left pane, you will see a *Call Stack* tab. While yellow highlighted line in the editor shows you where you currently are (should be inside the `f1()` function), the *Call Stack* shows you how did you get where. In this case it should show:

f1	test01.py	2:1
f2	test01.py	7:1
<module>	test01.py	9:1

The calls are stacked from bottom to top, so this means that a function was called in the main module in line 9, you ended up in function `f2` in line 7, and then in function `f1` and in line 2. Experiment with stepping in and out of functions while keeping an eye on this. You might not need this information frequently but could be useful in out later projects with multiple nested function calls.

7.7   Documenting your function

Writing a function is only half the job. You need to document it! Remember, this is a good habit that makes your code easy to use and reuse. There are different ways to document the code but we will use NumPy docstring convention. Here is an example of such documented function

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

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

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

Take the look at the manual and document the `input_response` function. You will not need the `Parameters` section as it currently accepts no inputs.

Update your code in `code02.py`.

7.8   Using prompt

In the future, we will be asking about a specific number that is a current guess by the computer, thus we cannot use a fixed prompt message. Modify the `input_response` function by adding a `guess` parameter. Then, modify the prompt that you used for the input() to include that number. Update functions' documentation. Test it by calling with different values for the `guess` parameter.

Put your code into `code03.py`.

7.9   Splitting interval in the middle

Let us practice writing functions a bit more. Recall that the computer should use the middle of the interval as a guess. Create a function (let us call it `split_interval()` or something like that) that takes two parameters — `lower_limit` and `upper_limit` — and returns *an integer* that is closest to the middle of the interval. The only tricky part is how you convert a potentially float number (e.g, when you are trying to find it for the interval 1..10) to an

integer. You can use function int() for that. However, read the documentation carefully, as it *does not* perform a proper rounding (what does it do? read the docs!). Thus, you should round() the number to the closest integer before converting it.

Write a function, document it, and test it by checking that numbers are correct.

Put you `split_interval()` function and the testing code into `code04.py`.

7.10 Single round

You have both functions that you need, so let us write the code to initialize the game and play a single round. The initialization boils down to creating two variables that correspond to the lower and upper limits of the game range (we used 1 to 10 so far, but you can always change that). Next, the computer should generate a guess (you have your `split_interval()` function for that) and ask the player about the guess (that is the `input_response()` function). Once you have the response (stored in a separate variable, think of the name yourself), you can update your upper or lower limit using an if..elif..else statement. Print out a joyous message, if computer's guess was correct.

Put both functions and the script code into `code05.py`.

7.11 Multiple rounds

Extend the game, so that the computer keeps guessing until it finally wins. You already know how to use the while loop, just think how you can use participant's response as a loop condition variable. Also, think about the initial value of that variable and how to use it so you call `input_response()` only at one location.

Put the updated code into `code06.py`.

7.12 Playing again

Modify the code, so that you can play this game several times. You already know how to do this and the only thing you need to consider is where exactly should you perform initialization before each round. As you already implemented that for the last game, you might be tempted to look how you did it or, even, copy-paste the code. However, I would recommend writing it from scratch. Remember, your aim is not to write a program but to learn how to do this and, therefore, the journey is more important than a destination.

Put the updated code into `code07.py`.

7.13 Best score

Add the code to count the number of attempts that the computer required in each round and report the best score (fewest number of attempts) after the game is over. You will need one variable to count the number of attempts and one to keep the best score. Again, try writing it without looking at your previous game.

Put the updated code into `code08.py`.

7.14 Using you own libraries

You already know how to use existing libraries but you can also create and use your own. Take the two functions that you developed and put them into a new file called `utils.py` (do not forget to put a multiline comment at the top of the file to remind you what is inside!) . Copy the remaining code (the global script) into `code09.py`. It will not work in its current state as it won't find the two functions (try it to see the error message), so you need to import from your own `utils` module. Importing works exactly the same way as for other libraries. Note that even though your file is `utils.py`, the module name is `utils` (without the extension).

Put function into `utils.py`, the remaining code into `code09.py`.

7.15 Ordnung muss sein!

So far, you only imported one library at most. However, as Python is highly modular, it is very common to have many imports in a single file. There are several rules that make it easier to track the imports. When you import libraries, all import statements should be at the top of your file and you should avoid putting them in random order. The recommended order is 1) system libraries, like `os` or `random`; 2) third-party libraries, like `psychopy`; 3) your project modules . 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.

7.16 Putting video into videogames

Submit your files and be ready for more excitement as we are moving onto
"proper" videogames with PsychoPy.

Chapter 8

Gettings started 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 currently the best Python library for developing psychophysical experiments (and this is what we are after).

8.1 Chapter concepts

- Understanding how to use classes and objects.
- Using named parameters in functions.
- Understanding PsychoPy units system.
- Using basic Psycho visual stimuli and handling user inputs.

8.2 Minimal PsychoPy code

Copy-paste the following code into `code01.py` file (you did remember to create a new folder for the chapter?):

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


# this imports two modules from psychopy
# visual has all the visual stimuli, including the Window class
```

```python
# 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. If you get an error saying that `psychopy` library is not found, check the active Python interpreter. 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 a key press) and it should close. Not very exciting but does show that everything works as it should.

Put your code into *code01.py*.

The code is simple but packs quite a few novel bits. First line is easy, we simply import visual and event modules from *psychopy* library (a library can be itself organized into sublibraries to make things even more modular). Then, we create an *object* `win` using a *class* Window with custom size. Third line uses function waitKeys() from *event* module to wait for a key press. The last one closes the window by calling its `close` *method*. You should have little trouble with lines #1 and #3 but you need to learn about object-oriented programming to understand #2 and #4.

8.3   Classes and objects

The PsychoPy library is collection of *classes* that you use to create *objects*, an approach called *object-oriented programming*. The core idea is in the name: Instead of keeping variables (data) separate from functions (actions), you combine them in an object that has attributes[1] (its own variables) and methods (its own functions). This approach utilizes our natural tendency to perceive the world as a collection of interacting objects.

First, you need to understand an important distinction between *classes* and *objects*. A *class* is a "blueprint" that describes properties and behavior of *all* objects of that class. This "blueprint" is used to create an *instance* of that class, which is called an *object*. For example, Homo sapiens is a *class* that describes

---

[1]Also called properties

species that have certain properties, such as height, and can do certain things, such as running. However, Homo sapiens as a class has only 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 performing 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) have different height and a different (typically better) ability to run. Thus, a class describes all common properties and methods that all *instances* of the class (all objects) will have. But individual object will behave differently because of different values of their properties. This means that whenever you meet a Homo sapien, you could be sure that they have height per se but will need to look at an individual *instance* to figure what height they have.

`Window` is a class that describes properties that a PsychoPy window must have and actions it can perform (you can see the complete list in the manual). To create an object, we use its class definition and store the result in a variable. In the code above we call `Window` class[2] while passing custom parameters to it (`size=(800, 600)`) and store an object that it returns in variable `win`.

Attributes are, essentially, variables that belong to the class and, therefore, variables that each object will possess. For example, a `Window` class has size attribute that determines its on-screen size in pixels. It also has (background) color, an attribute that determines whether it should be shown in full screen mode, etc. Thus, a `win` object will have all these attributes and they will have specific values.

To understand both properties and class/object distinction better, put a breakpoint on the third line of code (`event.waitKeys()`) and fire up the debugger via **F5**. Once the window is created, the execution will pause and you will find a `win` object in *Variables/Locals*. Click on it and it will expand to show all attributes and its values, including `size` (check that it is `[800, 600]`). Note that you will not see `Window` itself in the same list. This is because it is a class, an abstract concept, whereas as `win` is its instance and object of that class.

Methods, such as `Window.close()` are, essentially, functions that belong to the class/object and perform certain actions on the object. For example, method close() closes the window, flip() updates it after we finished drawing in it, etc. What is important is to remember is that each method will act only on the object *it belongs to* and not on other instances of the same class. This means that you can create two windows (`win1` and `win2`) and calling `win1.close()` will close the first but not the second window (try this out!). Same goes for attributes, changing them in one object will not affect any other objects of the

---

[2]Technically, we call a class constructor method called `__init__` but this is not important for now.

same class, just like changing a value in one variable will not affect the other ones.

Although we barely scratch the surface of object-oriented programming, it will be enough for us to be able to use classes defined for us in PsychoPy library.

8.4   Function parameters: default values, passing by position or by name

There are a few more curious bits in the `visual.Window(size=(800, 600))` call above that we need to discuss. These curiosities are related to functions (and, therefore, methods) not classes per se. First, constructor method of the Window class has a lot of arguments (when we construct an object, we call a constructor *method* of the class, which is why we are talking about functions). And yet, we only passed one of them. This is because you can specify default values for individual parameters. In this case, if a parameter is omitted, a default value is used instead

```python
def divide(x1, x2=2):
    """
    Divides numbers, uses 2 as a second value if a second term is omitted.

    Parameters:
    ----------
    x1 : number
    x2 : number, defaults to 2

    Returns:
    ----------
    number
    """
    return x1 / x2
print(divide(2))
#> 1.0
print(divide(2, 4))
#> 0.5
```

If you look at documentation, you will see that for the Window class constructor *all* parameters have a default value. This is part of PsychoPy's philosophy of combining rich customization (just look at the sheer number of parameters!) with simplicity of use through sensible defaults (specify nothing and the window will still work).

Second, we did not just pass the value but specified which parameter this value is for via `size=(800, 600)`. This notation is called keyword arguments. The

advantage is in making it more explicit which parameter you are passing a value through. Plus, it allows you to put parameters any order, if that is more relevant given the context[3]. If you do not use names, the values are assigned to individual parameters based on their *position* (a.k.a. positional parameters). You can even mix the two, but positional parameters must come first, see documentation if you want to know more.

```python
# using positional parameters
print(divide(2, 4))
#> 0.5
```

```python
# using keyword arguments
print(divide(x2=4, x1=2))
#> 0.5
```

```python
# mixing positional and keyword arguments
print(divide(2, x2=4))
#> 0.5
```

8.5 Adding main loop

Currently, not much is happening in our program. One thing we need to add is a loop in which we can repeatedly draw in a window (and update it via its flip() method), check user input, and perform any other necessary actions.

First, let us add the loop and handling of user inputs (the fun drawing part will be next). The loop goes between opening and closing the window:

```
importing libraries
opening the window

--> our main loop <--

closing the window
```

The loop should be repeated until the user presses an *escape* key and, therefore, you will need a variable that signals this. My approach is to create a variable `gameover` initializing it to `False` and repeat the loop as long as the game not over. Then, in 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 `keyList` were pressed. Store that returned

---

[3]However, stick to original order for consistency otherwise.

value in a temporary variable (I tend to call it `keys`). You will learn about lists only in the *next* chapter, so for now use a ready-made: `len(keys) > 0` is a comparison that is `True` if list is not empty. If the list is indeed not empty, that means that the user pressed *escape* (as that is the only key that we specified in the function call) and the game should be over. Think how can you do it *without* an `if` statement, computing the logical value directly?

Put your code into *code02.py*.

8.6   Adding text message

Although we are now running a nice game 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) before the main loop. There are quite a few parameters that you can play with but minimally, you need to pass the window the text should be displayed in (our `win` variable) and the actual text you want to display (`text="Press escape to exit"`). For all other settings PsychoPy will use its defaults (default font family, color and size, placed right at the windows' 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 and then update the window by *flipping*[4] it. Note that you call flip() only *once* after *all* stimuli are drawn. I typically organize this code into a separate chunk and prepend it with a comment line `# drawing stimuli`.

The `# drawing stimuli` chunk goes inside the main loop either before[5] or after the keyboard check. Organize the latter also as a separate code chunk with its own brief comment.

Put your code into *code03.py*.

Now, you should have a nice, although static, message positioned at the window's center 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 class 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.

---

[4]This is called flipping because a window has two buffers: one that is currently displayed on the screen and the other one in which you can draw your stimuli. Once you are done with drawing, you "flip" the buffers so that they exchange their places. Now the one you drew in gets displayed and you have the other buffer to draw in.

[5]my personal preference but in most cases it makes no difference

8.7 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 just used) has pos property that specifies (you guessed it!) its position within a window. However, to make your life easier, PsychoPy first complicates it by having **five** (5!) different position 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 those 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 outline 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)`. However, on *some* computers a curious bug prevents PsychoPy from drawing the outline correctly. If you end up staring at an empty screen, add `fillColor="white"` to the call and you should see a filled white square.

You draw the square just like you drew the text stimulus, via its draw() method. Create the code (either keep the text and draw both, or drop the text), run it to see a very white square.

Put your code into *code04.py*.

What? Your square is not really a square? Well, I've warned you: **Five** units systems!

8.8 Five units systems

8.8.1 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, horizontal limits depend on the aspect ratio. For our 800×600 window (4:3 aspect ratio), it will go from -0.666 to 0.666 (the window is 1.3333 window heights wide). For a 600×800 window (3:4 aspect ratio) from -0.375 to 0.375 (the window is 0.75 window heights wide), for a square window 600×600 (aspect ratio 1:1) 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, height units make *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>)`.

Put your code into *code05.py*.

By the way, which way is up: When y is below or above zero? 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 (this could be very confusing, if you forget about this when overlaying gaze data on an image you used in the study 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 *code06.py*.

8.8.2   Normalized units

Normalized units are default units 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 your white square stimulus, so it does look like a square.

Put your code into *code07.py*.

8.8.3   Pixels on screen

For   pixels   on   screen   (https://psychopy.org/general/units.html#pixels-on-screen) units, 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 has changed or you are using a full screen window on a monitor with an unknown resolution. In short, you should use them only if they dramatically simplify your code.

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

Put your code into *code08.py*.

8.8.4 Degrees of visual angle

Unlike the three units above, using degrees of visual angle requires you knowing a physical size of the screen, its resolution, and 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.

8.8.5 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 usage cases[6].

8.9 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 and generate a new position using random.uniform(a, b) function, that generates a random value within *a..b* range[7]. Generate two values (one for x, one for y). If you use `"norm"` units, your range is the same (from -1 to 1) for the two dimensions. However, if you used `"height"` units, you need to take into account the aspect ratio of your window (4:3 if you are using 800×600 pix window).

Put your code into *code09.py*.

8.10 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.

---

[6]So specific that I cannot think of one, to be honest.

[7]You need to import the random library for this, of course.

So far, we restricted it to just **escape** button and checked whether any (hence, **escape**) button was pressed.

You will learn about lists and indexes in the next chapter, so here is another ready-made. First, add `"space"` to the `keyList` parameter. Next, use conditional if statement to check whether event.getKeys() returned a key press. If it did (`len(keys) > 0`), you can now check whether `keys[0]` is equal to `"space"` or `"escape"`[8]. If it was the latter, the game is over as before. If it was `"space"` then move the square to a new random position (and do not move it on every frame!)

Hint, if you are debugging

Put your code into *code10.py.*

8.11   Basics covered

There is plenty more to learn about PsychoPy but we've got the basics covered. Submit your files and get ready to Whack a Mole!

---

[8]You can use `if..else`, because we only have two options but I would recommend to go for a more general solution `if..elif`

Chapter 9

Whack-a-Mole

Today you will create your first *video* game Whack-a-Mole. The game itself is very much a reaction time experiment: moles/targets appear after a random delay at one of the predefined locations, the player's task is to whack (press a corresponding button) the mole/target before it disappears. Your final game should look approximately like the one in the video: Circles (moles) turn white, if I hit the correct button in time.

9.1 Chapter concepts

- Storing many items in lists.
- Iterating over items use for loop.
- Generating a list of number using range().
- Making a pause and limiting time you wait for a key.

9.2 Lists

So far, we were using variables to store single values: computer's pick, player's guess, number of attempts, PsychoPy window object, etc. But sometimes we need to handle more than one value. We already had this problem in the computer-based Guess-the-Number game when we needed to store the remaining number range. We got away by using two variables, one for the lower and one for the upper limit. However, this approach clearly does not scale well and, sometimes, we might not even know how many values we will need to store. Python's lists are the solution to the problem.

A list is a mutable[1] sequence of items where individual elements can be accessed

---

[1]More on that and tuples (list's immutable cousins) later.

via their zero-based index. Extending the idea of variable-as-a-box, you can think about lists as a box with numbered slots. To store and retrieve a particular piece you will need to know both the *variable name* and the *index of the item* you are interested in within that box. Then, you work with a variable-plus-index in exactly the same way you work with a normal variable, accessing or changing its value via the same syntax as before.

A list is defined via square brackets `<variable> = [<value1>, <value2>, ... <valueN>]`. An individual slot within a list is also accessed via square brackets `<variable>[<index>]` where index is, again, **zero-based**[2]. This means that the *first* items is `variable[0]` and, if there are *N* items in the list, the last one is `variable[N-1]`. You can figure out the total number of items in a list by getting its length via a special len() function. Thus, you can access the last item via `variable[len(variable)-1]`[3]. Note the `-1`: If you list has 3 items, the index of the last one is 2, if it has 100, then 99, etc. I am spending so much time on this because it is a fairly common source of confusion.

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

Lists also allow you access more than one slot/index at a time via *slicing.* You can specify index of elements via `<start>:<stop>` notation. For example, `x[1:3]` will give you access to two items with indexes 1 and 2. Yes, *two* items: Slicing index goes from the `start` up to **but not including** the `stop`. Thus, if you want to get *all* the items of a list, you will need to write `x[0:length(x)]` and, yet, to get the last item alone you still write `x[len(x)-1]`. Confusing? I think so. I understand the logic but I find this stop-is-not-included to be counterintuitive and I still have to consciously remind myself about this. Unfortunately, this is a standard way to define sequences of numbers in Python, so you need to memorize this.

Do exercise #2 to build the intuition.

When slicing, you can 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>)` (so, last item is included). 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)]`.[4]

Do exercise #3.

You can also use *negative* indexes that are computed relative to length of the list. For example, if you want to get the *last* element of the list, you can say `my_pretty_numbers[len(my_pretty_numbers)-1]`

---

[2]This is typical for "classic" programming languages but less so for ones that are linear algebra / data science oriented. Both Matlab and R use one-based indexing, so you need to be careful and double-check whether you are using correct indexes.

[3]There is a simpler way to do this, which you will learn in a little while.

[4]Note, that this is almost but not quite the same thing as just writing `my_pretty_numbers`, the difference is subtle but important. We will return to it later when talking about mutable versus immutable types.

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 the *including the start but excluding the stop* catch: `my_pretty_numbers[:-1]` will return all but last element of the list not the entire list!

Do exercise #4.

Slicing can be extended by specifying a `step` via `stop:start:step` notation. `step` can be negative, allowing you to build indexes in the reverse order:

```
my_pretty_numbers = [1, 2, 3, 4, 5, 6, 7]
my_pretty_numbers[4:0:-1]
#> [5, 4, 3, 2]
```

However, you must pay attention to the sign of the step. If it goes in the wrong direction then `stop` cannot be reached, Python will return an empty list.

```
my_pretty_numbers = [1, 2, 3, 4, 5, 6, 7]
my_pretty_numbers[4:0:1]
#> []
```

Steps 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.

If you try to to access indexes *outside* of a valid range, Python will raise an IndexError[5]. Thus, trying to get 6th element (index 5) of a five-element-long list will generate a simple and straightforward error. However, if your *slice* is larger than the range, it will be truncated without an extra warning or an error. So, for a five-element list `my_pretty_numbers[:6]` or `my_pretty_numbers[:600]` will both return all numbers (effectively, this is equivalent to `my_pretty_numbers[:]`). Moreover, if the slice is empty (`2:2`, cannot include 2 because it is a stop value, even though it starts from 2 as well) or the entire slice is outside of the range, Python will return an empty list, again, neither warning or error is generated.

Do exercise #6.

---

[5]If you are familiar with R and its liberal attitude towards indexes, you will find this very satisfying.

In Python lists are dynamic, so you can always add or remove elements to it, see [the list of methods]](https://docs.python.org/3/tutorial/datastructures. html#more-on-lists). You can add a new item to the of the end of the list via `append(<new_value>)` method

```python
my_pretty_numbers = [1, 2, 3, 4, 5, 6, 7]
my_pretty_numbers.append(10)
my_pretty_numbers
#> [1, 2, 3, 4, 5, 6, 7, 10]
```

Or, you can `insert(<index>, <new_value>)` *before* an element with that index. Unfortunately, this means that you can use an arbitrary large index and it will insert a new value as a *last* element without generating an error.

```python
my_pretty_numbers = [1, 2, 3, 4, 5, 6, 7]
my_pretty_numbers.insert(2, 10)
my_pretty_numbers.insert(500, 20)
my_pretty_numbers
#> [1, 2, 10, 3, 4, 5, 6, 7, 20]
```

You can remove an item using its index via `pop(<index>)`, note that the item is *returned* as well. If you omit the index, `pop()` removes the *last* element of the list. Here, you can only use valid indexes.

```python
my_pretty_numbers = [1, 2, 3, 4, 5, 6, 7]
my_pretty_numbers.pop(-1)
#> 7
my_pretty_numbers.pop(3)
#> 4
my_pretty_numbers
#> [1, 2, 3, 5, 6]
```

Do exercise #7.

9.3   Basic game scaffolding

Phew that was *a lot* about lists[6]. However, all work and no play makes Jack a dull boy! So let us start with a basic PsychoPy scaffolding. Here the code structure:

---

[6]and we barely scratched the surface!

```
import libraries from [psychopy]
create the PsychoPy window (visual.Window())
flip the window (.flip())
wait for a player to press the escape key (event.waitKeys())
close the window (.close())
```

Try doing it from scratch. I have left hints to help you with this and you can always consult the online documentation. Do not forget to document the file and to split your code into meaningful chunks with comments (if needed).

Put your code into *code01.py*.

9.4 Three moles

Let us create three moles that will be represented by circles. Create a new list variable `moles` and put three circles into it. One should go to the left, one dead center, and one to the right. Watch a video above to see what I mean. Think of a reasonable size (which units make keeping circle a circle easier?) and position. You can also use different colors for them, as I did).

You can either create an empty list and then `append()` circles one at a time. Or you can use square brackets to put all three of them into the list in one go. Then draw() circles before you flip the window and wait for a key press. Note that you have to draw them one at a time. Therefore, you will need to add three lines for this but the next section will show you an easier way.

Put your code into *code02.py*.

9.5 For loop

In the code above, we needed to iterate over three moles (circles) that we had in a list. Python has a tool just for that: a for loop that iterates over the items in any sequence (our list is a sequence!). Here is an example:

```
numbers = [2, 4, 42]
for a_number in numbers:
    print("Value of a_number variable on this iteration is %d"%(a_number))
    a_number = a_number + 3
    print("  Now we incremented it by 3: %d"%(a_number))
    print("  Now we use in a formula a_number / 10: %g"%(a_number / 10))
#> Value of a_number variable on this iteration is 2
#>   Now we incremented it by 3: 5
#>   Now we use in a formula a_number / 10: 0.5
```

```
#> Value of a_number variable on this iteration is 4
#>   Now we incremented it by 3: 7
#>   Now we use in a formula a_number / 10: 0.7
#> Value of a_number variable on this iteration is 42
#>   Now we incremented it by 3: 45
#>   Now we use in a formula a_number / 10: 4.5
```

Here, the code inside the `for` loop is repeated three times because there are three items in the list. On each iteration, next value from the list gets assigned to a temporary variable `a_number` (see the output). Once the value is assigned to a variable, you can use it just like any variable. You can print it out (first `print`), you can modify it (second line within the loop), use its value for when calling other functions, etc. To better appreciate this, copy-paste this code into a temporary file (call it `test01.py`), put a breakpoint onto the first `print` statement and then use **F10** to step through the loop and see how value of `a_number` variable changes on each iteration and then it gets modified in the second line within the loop.

Note that you can use the same break statement as for the while loop.

Do exercise #8.

9.6   Drawing in a loop

Now that you have learned about the for loop, it is easy to draw the moles. Just iterate over the list (come up with a good temporary variable name) and `draw()` a current item (which is in your temporary variable).

Put your code into *code03.py*.

9.7   range() function: Repeating code N times

Sometimes, you might need to repeat the code several times. For example, imagine that you have 40 trials in an experiment. Thus, you need to repeat a trial-related code 40 times. You can, of course, build a list 40 items long by hand and iterate over it but Python has a handy range() function for that. `range(N)` yields N integers from 0 to N-1 (same up-to-but-not-including rule as for slicing) that you can iterate over in a for loop.

```python
for x in range(3):
    print("Value of x is %d"%(x))
#> Value of x is 0
#> Value of x is 1
#> Value of x is 2
```

You can modify range() function behavior by providing a starting value and a step size. But in its simplest form `range(N)` is a handy tool to repeat the code that many times. Note that while you always need to have a temporary variable in a `for` loop, sometimes you may not use it at all. In cases like this, you should use `_` (underscore symbol) as a variable name to indicate the lack of use.

```python
for _ in range(2):
    print("I will be repeated twice!")
#> I will be repeated twice!
#> I will be repeated twice!
```

Alternatively, you can use `range()` to loop through indexes of a list (remember, you can always access an individual list item via `var[index]`). Do exactly that[7]! Modify your code to use range() function in the for loop (how can you compute the number of iterations from the length of the list?), use temporary variable as an index for the list to draw each item[8]. When in doubt, put a breakpoint inside (or just before) the loop and step through your code to understand what values a temporary loop variable gets and how it is used.

Put your modified code into *code04.py*.

9.8 A random mole

Drawing all three moles served as a practical exercise with loops but in a real game we need to shown only one random target at a time. We could create the three targets as before and draw one of them. However, later on we would like to change the color of the target to indicate that the player did hit it, so it is simpler (if a bit wasteful) to create a single mole every time we need one.

For this, define one CONSTANT with a list of three colors that you used and another one with three horizontal locations (the vertical location is the same, so we do not need to worry about it). Next, randomly pick which target out of three you want to create, i.e., we need to generate an *index* of the target. You can do it either via random.randrange() or via random.choice() building the range yourself via the function with the same name you have just learned about (remember to organize your imports alphabetically). Store the index in a variable with a meaningful name[9] and use it with constants to create the target of the corresponding color at a corresponding location. Then, you need to draw that single target before waiting for a key press.

---

[7]Note, this is note a *better* way but an *alternative* way to do this.

[8]Style hint: if a variable is an *index* of something, I tend to call it `isomething`. E.g., if it holds an index to a current mole, I would call it `imole`. This is *my* way of doing it. Others use `i_` prefix or an `_i` suffix. But either way, it is a useful naming convention. Remember, the easier it is to understand the meaning of a variable from its name, the easier it is for you to read and modify the code.

[9]`itarget`? `imole`?

Once you have the code, put a breakpoint and check that the value of the index variable matches what is shown on a screen[10].

Put your modified code into *code05.py*.

9.9   Random time

What makes Whack-a-Mole game fun is not only that you do not know *which* mole will appear but you also do not know *when* it will appear and *how much time* you have to whack it. Thus, we need to modify our presentation schedule. We need a blank period of a random duration (I would suggest between 0.75 s to 1.5 s) and limited presentation duration (between 0.5 to 0.75 s). First, you need to define these ranges as constants. Now that you know lists you can use a single variable to hold both ends of the range. Then, you need to generate two numbers (one for the blank another for the presentation) coming from a uniform distrubition within that range. Finally, you need to time your blank and presentation using the wait() function from the clock module.

Now is time to update and structure you code. Here is a approximate outline (note that I have dropped the wait for keys):

```
"""Document your file
"""
import all libaries you need in an alphabetical order

define CONSTANTS

create window

# generating random parameters for the trial
pick random index for the mole
create the mole
generate random durations for blank and presentation interval

# blank
clear window (win.flip() alone)
wait for "blank duration" seconds

# presentation
draw the mole
wait for "presentation duration" seconds
```

---

[10]I know it feels redundant but these are little checks that cost little time by themselves but help you avoid wasting lots of time on tracing weird mistakes. Here, you check that your expectations (if the middle target is shown, the index should be 1) match the reality. Once you check this, you do not *expect* it to be true, you *know* it to be true!

```
close the window
```

Note that it has no response processing at the moment and that window should close right after the stimulus is presented.

Put your code into *code06.py*.

9.10   Repeating trials

You already know how to repeat the same code many times. Decide on number of trials / rounds (define this as a constant) and repeat the single round that many times. Think about what code goes inside the loop and what should stay outside for the randomization to work properly.

Put your code into *code07.py*.

9.11   Exit strategy

I hope that you used a small number of trials because (on my advice, yes!) we did not program a possibility to exit the game via the **escape** key. To put it in, we will replace *both* wait() calls with waitKeys() function. It has `maxWait` parameter that by default is set to infinity but can be set to the duration we require. If a player does not press a key, it will work just like wait() did. If a player presses a key (allow only `"escape"` for now), it means that they want to abort the game (the only possible action at the moment). Thus, assign the returned value to a temporary variable (`keys`?) and check whether it is equal to `None`[11]. If it is not equal to `None`, break out of the loop!

Put your code into *code08.py*.

9.12   Whacking that mole

We have moles that appear at a random location after a random delay for a random period of time. Now we just need to add an ability to whack 'em! You whack a mole only when it is present. Thus, we only need to modify and handle the waitKeys() call for the presentation interval.

First, create a new constant with three keys that correspond to three locations. I would suggest using `["left", "down", "right"]`, which are cursor keys[12].

---

[11]Confusingly, if no key was pressed, getKeys() returns an empty list but waitKeys() returns `None` and `None` has no length.

[12]Want to know key codes for sure? Write a small program that open a window and then repeatedly waits for any key press and prints out into console.

Next, you need to use them for the `keyList` parameter. However, we cannot use this list directly, as we also need the **escape** key. The simplest way is to put "escape" into its own list and concatenate the two lists via `+`: `["escape"]` `+ YOUR_CONSTANT_WITH_KEYS`. Do this concatenation directly when you set a value to the `keyList` in the function call. Before we continue, run the code and test that you can abort the program during the presentation (but not during the blank interval) by pressing any of these three keys. Also check that **escape** still works!

Now that we have keys to press, we need more sophisticated processing (we gonna have quite a few nested conditional statements). We still need to check whether waitKeys() returned `None` first. If it did not, it must have returned a list of pressed keys. Actually, it will be a list with just a single item[13], so we can work with it directly via `keys[0]`. Use conditional if-else statement to break out of the loop if the player pressed **escape**. Otherwise, it was one of the three "whack" keys.

Our next step is to establish which index the key corresponds to. Python makes it extremely easy as lists have `.index(value)` method that returns the index of the value within the list. You have the (CONSTANT) list with the keys, you have the pressed key: Figure out the index and check whether it matches the index of the target (`imole` variable in my code). If it does, let us provide a visual feedback of success: change mole (circle) `fillColor` to white, draw it, and wait for 300 ms (setup a constant for feedback duration). This way, the mole will turn white and remain briefly on the screen when hit but will disappear immediately, if you missed.

Put your code into *code09.py*.

9.13   You did it!

Congratulations on your first video game! It could use some bells-and-whistles like having a score, combos would be cool, proper mole images instead circle, etc. but it works and it is fun (if you do not feel challenged, reduce the presentation time)! Submit your files and next time we will ditch the keyboard and learn how to handle the mouse for the Memory game.

---

[13]You will get more than one item in that list only if you set `clearEvents=False`. In this case, you will get the list of keys pressed before the call. However, if you opted for a default `clearEvents=True`, you will get only one key press in the list (at least I was never able to get more than one).

Chapter 10

Memory game

Today, you will write a good old *Memory* game: Eight 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 for the game 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.

10.1   Chapter concepts

- Mutable vs. immutable objects
- Showing images.
- Working with files via os library.
- Using other dictionary containers.
- List operations.
- Looping over both index and item via list enumeration.

10.2   Variables as Boxes (immutable objects)

In this game, you will use dictionaries. These are *mutable*, like lists in constrant to "normal" *immutable* values (integers, floats, strings). You need to learn about this distinction as these two kinds of objects (values) behave very differently under some circumstances, which is both good (power!) and bad (weird unexpected behavior!) news.

---

[1]The images are courtesy of Kevin David Pointon and were downloaded from OpenClipart. They are public domain and can be used and distributed freely.

You may remember the *variable-as-a-box* metaphor that I used it to introduce variables. In short, a variable can be thought of as a "box" with a variable name written on it and a value being stored "inside". When you use this value or assign it to a different variable, you can assume that Python *makes a copy* of it[2] and puts that *copy* into a different variable "box". When you *replace* value of a variable, you take out the old value, destroy it (by throwing it into a nearest black hole, I assume), create a new one, and put it into the variable "box". When you *change* 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 the new one back into the variable "box". The important point is that although a *variable* can have different immutable values (we changed `imole` variable on every round), the immutable *value* itself never changes. It gets *replaced* with another immutable value but *never changes*[3].

The box metaphor explains why the scopes work the way they do. Each scope has its own set of boxes and whenever you pass information between scopes, e.g., from a global script to a function, a copy of a value (from a variable) is created and put into a new box (e.g., a parameter) inside the function. When a function returns a value, it is copied and put in one of the boxes in the global script (variable you assigned the returned value to), etc.

However, this is true only for *immutable* objects (values) such as numbers, strings, logical values, etc. 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.

10.3   Variables as post-it stickers (mutable objects)

Mutable objects are lists, dictionaries[4], and classes, i.e., things that can change. The key 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[5]. For example, you can add elements to your

---

[2]Not really, but this makes it easier to understand.

[3]A metaphor attempt: You can wear different shirts, so your *look* (variable) changes but each individual shirt (potential values) remains the same (we ignore the wear and tear here) irrespective of whether your are wearing it (value is assigned to a variable) or not.

[4]Coming up shortly!

[5]Building on the looks metaphor: You can change your look by using a different (immutable) shirt or by *changing* your haircut. You hair is mutable, you do not wear a different one on different days to look different, you need to modify it to look different.

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 when you assign them to a different variable or use as a parameter value 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 a sticker (one of many) for a mutable object, try figuring out what will be the output below:

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

Do exercise #1.

Huh? That is precisely what I meant with "stickers on the same object". First, we create a list and put an x sticker on it. Then, we assign y *to the same list*, in other words, we put a y sticker on the same list. 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, a.k.a. "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.

```python
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 yet 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 and learned about scopes!), 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[6] 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.

10.4   Tuple: a frozen list (#tuple)

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 variable with 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 Jupyter Notebook)

```
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

---

[6]Well, at least me!

```python
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 and you want them to have no link to the original variable, 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?

```python
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 are the same or 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. This way you can two *different* lists with *identical* values. 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 immutable/mutable, tuple/list, copy/reference confusion, you are just being a normal human being. I understand the (computational) reasons for doing things this way, I am aware of this difference and how useful this can be but it still catches me by surprise from time to time!

10.5   Minimal code

Enough of theory, let us get busy writing the game. As usual, let us start with a minimal code (try doing it from scratch instead of copy-pasting from the last game):

```
importing psychopy modules that we need

creating a window of a useful size and useful units

waiting for a key press

closing the window
```

The first thing you need to decide on is the window size *in pixels* and which units would sizing and placing cards easier. Each chicken image is 240×400 pixels and, for the game, we need place for *exactly* 4×2 images, i.e. our window must be 4 cards wide and 2 cards high. Do not forget to document the file!

Put your code into `code01.py`.

10.6   Drawing an image

We used (abstracta and boring) circles to represent moles but today we will use actual images of chicken (see instructions above on downloading them). Using an image stimulus in PsychoPy is very straightforward because it behaves very mcu like other visual stimuli you already know. First, you need to create an new object by calling `visual.ImageStim(...)`. You can find the complete list of parameters in the documentation 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"` (images are in a subfolder and therefore we need to use a relative path).
- size: `size=(???, ???)`. That is one for you to compute. If you picked norm units, as I did, then window is 2 units wide and 2 units high but for height it is 1 units height and *aspect-ratio* units wide. We want to have a 4×2 images, what is the size (both width and height) of each image in the units of your choice?

Draw chicken image (it should appear at the center of the screen).

Put your code into `code02.py`.

10.7   Placing an image (index to position)

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 (which is also, typically, at the center of the screen). However, we will need to draw eight images, each at its designated location. You need to create a function that takes an image index (it goes 0 to 7) and returns a list with pair of values with its location on the screen. Below is a sketch of how index correspond to the location. Note that image location (pos corresponds to the *center* of the image.
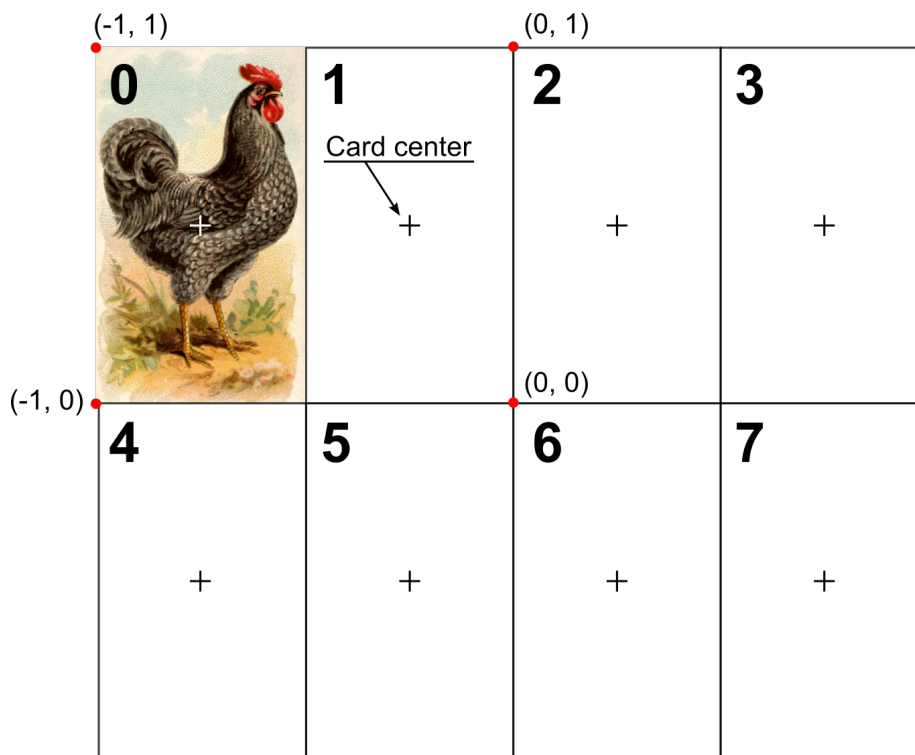
Figure 10.1: Card location index

Name the function `position_from_index`. It should take one argument (`index`) and return a list with (`<x>`, `<y>`) coordinates in the PsychoPy units (from now on I assume that these are norm). You can then use this value for the pos argument of the ImageStim().

The computation might look complicated, so let me get you started. How can you compute *x* coordinate for the *top* row? Concentrating on the top row alone makes things simpler because here the *column index* is the same as the overall index: The left-most column is 0, the next one is 1, etc. You need a simple

algebra of $x = a_x + b_x \cdot column$. You can easily deduce out both a_x and b_x if you figure out locations of the first and second cards by hand. Same goes for the y coordinate. Assuming that you know the *row*, which is either 0 (top row) or 1 (bottom row), you can compute $y = a_y + b_y \cdot row$.

But, I hear you say, you do not have row and column indexes, only the overall index! To compute those you only need to keep in mind that each row has *four* cards. Then, you can make use of two special division operators: 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`.

My suggestion would be first to play with individual formulas in Jupyter Notebook, which makes it easier to try out (dividing) things and seeing the result, putting various values into formulas, etc. Once you are confident that the code is working, turn it into a function, document it, and put into a separate file (*utilities.py*, do not forget to put a comment at the top of the file as well!). You can then import it in the main script and use it to place the card. Try out different indexes and make sure that the card appears where it should. Remember, put a breakpoint and step through the program while watching variables, if things do not work as you expected.

Put `position_from_index` into `utilities.py`.   Put update code into `code03.py`

10.8   Backside of the card

A chicken image is a card *face* but the game starts with the cards face down, so the player should see their backs. We will use a plain rectangle for as a backside. Pick a nice looking combination of `fillColor` (inside) and `lineColor` (contour) colors. Modify your code, to draw image (face 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 positioned correctly.

Put your code into `code04.py`.

10.9   Dictionaries

Each card that we use has plenty of properties: A front (image), a backside (rectangle), and will have other properties such as which side should be shown or whether card is already taken off the screen. This calls for a container, so we could put all these relevant bits into a single variable. We *could* put into a list and use numerical indexes to access individual elements (e.g., `card[0]` would

be front image but `card[2]` would indicate the active side) but indexes do not have meaning per se, so figuring out how `card[0]` is different from `card[2]` would be tricky. Python has a solution for cases like this: dictionaries.

A dictionary is a container that stores information using *key : value* pairs. This is similar to how you look up a meaning or a translation (value) of a word (key) in a 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"}
```

Once you created a dictionary, 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. This is similar to using the `list[index]` combination, the only difference is that `index` must be an integer, whereas `key` can be any hashable[7] value.

10.10 Using a dictionary to represent a card

Our card has the following properties, so these will be key-value entries in a dictionary

1. `"front"`: front side (image of a chicken).
2. `"back"`: back side (rectangle).
3. `"filename"`: identity on the card that we will use later to check whether the player opened two identical cards (their filenames match) or two different ones.
4. `"side"`: can be either `"front"` or `"back"`, information about which side is up (drawn on the screen). Set it `"back"` because, initially, all cards are face down. However, you can always set it temporarily to `"front"` to see how the cards are distributed.
5. `"show"`: a logical value, set it to `True`. We will use it later to mark out cards that are off the table and are, therefore, not shown. Initially, all cards are shown, so all cards should be created with `"show"` being equal to `True`.

---

[7]Immutable values are hashable, whereas mutable ones, like dictionaries and lists, are not. This is because mutable objects can *change* while the program is running and therefore are unusable as a key. I.e., it is hard to match by a key, if the key can be different by the time you need to access the dictionary.

Create a dictionary variable (name it `card`) and fill it with the relevant values (use either `"front"` and "`back`" for `"side"` key) and stimuli (you can put PsychoPy stimuli into a dictionary just like we put them into a list earlier). Modify you code so that it draws the correct image based on the value of the `"side"` entry. Note that you **do not need an if-statement for this**! Think about a key you need to access these two sides and the value that you have in for the `"side"` key.

Put your code into `code05.py`.

10.11   Card factory

You have the code to create one card but we need eight of them. This definitely calls for a function. Write a function (put it into `utilities.py` to declutter the main file) that takes three parameters

1. a window variable (you need it to create PsychoPy stimuli),
2. a filename,
3. card position index,

and returns a dictionary, just like the one you created. You very much have the code, you only need to wrap it into a function and document it. Call function `create_card` and use it in the main script to create `card` dictionary. Think about libraries you will now need import in *utilities.py*.

Put `create_card` into `utilities.py`. Put code into `code06.py`.

10.12   Getting a list of files

For a single card, we simply hard-coded the name of an 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. This is covered by os library that contains various utilities for working with your operating system and, in particular, with files and directories. Specifically, os.listdir(path=".") 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)[8].

Try this out in a Jupyter Notebook (do not forget to import the os library). You should get a list of 8 files that are coded as *[r/l][index].png*, where *r* or *l*

---

[8]Use absolute path only if it is the only option, as it will almost certainly will break your code on another machine.

denote a direction the chicken is looking. However, for our game we need only four images ($4 \times 2 = 8$ cards). Therefore, we need to select a subset of them, e.g., four random cards, chicken looking to the left or to the right only. Here, let us work with chicken looking to the left, meaning that we need to pick only files that start with "l". To make this filtering easier, we will use a cool Python trick called list comprehensions.

10.13   List comprehension

List comprehension provides 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[9]. 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 #7 in Jupyter notebook.

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

---

[9]A very arbitrary example!

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

Do exercise #8 in Jupyter notebook.

As noted above, you can also use a 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

```python
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:

```python
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 #9 in Jupyter notebook.

10.14   Getting list of relevant files

Use list comprehension to create a list of files of chicken looking left, *i.e.* with filenames that start with "l". Use .startswith() to check whether it starts with "l", store the list in `filenames` variable. Test you code in a Jupyter Notebook. You should get a list of four files.

10.14.1   List operations

Our list consists of four unique filenames but in the game each card should appear twice. There are several ways of duplicating lists. Here, We will use this as a opportunity to learn about list operations. Python lists implement two operations:

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

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

Note that this produces a *new* list and, therefore, that this is not equivalent to extend method `a.extend(b)`! The `+` creates a *new* list, `.extend()` extends the original list `a`.[10]

- List 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
#> [1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Use either operation or `.extend()` method to create the list where each filename is repeated twice. Hint, you can apply list multiplication directly to the filenames list you created via list comprehension (so, replicate it in that same line). Try this code out in a Jupyter Notebook.

10.15 Looping over both index and item via list enumeration

Now that we have a list of filenames, we can create a list of cards out of it. Our dictionary function requires both index and filename. The latter is the *item* of the list, the former is the *index* of that item. You could for a for loop by build the index using range() function but Python has a better solution for this: a enumerate() function! If, instead of iterating over a list, you iterate over enumerate(), you get a tuple[11] with both (`index`, `value`). Here is an example:

```
letters = ['a', 'b', 'c']
for index, letter in enumerate(letters):
    print('%d: %s'%(index, letter))
#> 0: a
#> 1: b
#> 2: c
```

---

[10]You will learn about practical implications of this later. For now, keep in mind that seemingly identical output might be fundamentally different underneath.

[11]A list you cannot change, more on this later.

And here is how you can use enumerate() for list comprehension.

```python
letters = ['a', 'b', 'c']
["%d: %s"%(index, letter) for index, letter in enumerate(letters)]
#> ['0: a', '1: b', '2: c']
```

10.16   Computing path

Originally, we specified image file name as `"Images/r01.png"`. This did the job but now we have many filenames that we need to join with the folder name to form a path string. On top of that, major operating systems disagree with Windows on where `/` (forward slash) or `\` (backslash) should be used for paths. To make your code platform-independent and, therefore, more robust, you need to construct a filename string using join function in path submodule. Thus, you can import *os* library and call it as `os.path.join(...)` (my personal preference). Or, you can use the same approach as for PsychoPy modules and import `path` from *os*, shortening the code. Or, of course, you can even import *join* directly but I find that lack of library information during use makes things harder to understand (even though the code is even shorter).

join takes path components as parameters 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 need to load multiple files, the *filename* part will vary. However, the *folder* where the images are located will be the same and, as per usual, it would a good idea to turn it into a formally declared CONSTANT.

Modify the `create_card` function so that it assumes that the `filename` parameter is just the filename with the folder name and, therefore, build the path by join it with the folder name (defined as a constant in the this module!). You now need to drop the `"Images/"` in the value that you pass to it. Test that the code works as before!

Update `create_card` in `utilities.py` Put updated code into `code07.py`.

10.17   A deck of cards

Let us put together all the code we need for figuring out cards' filenames, duplicating them, and creating the cards using filename and index.

Copy the code for building a duplicated list of filenames that you tested in Jupyter notebook to your main script (that'll be `code09.py`). Then, use enumerate and list comprehension over enumerated duplicate filenames to create

`cards` (plural, replacing your singular `card` variable) via `create_card` function you wrote earlier. Update your drawing code to loop over and draw all cards. If your default is `"side"` is `"back"`, things will look pretty boring. Change that to '"front"' for all cards to see their faces.

Put your code into `code08.py`.

10.18 Shuffling cards

When you draw cards faces, you will notice that duplicating filenames list produces a very orderly sequence that makes playing the game easy (and boring). We need to shuffle() the filename list *before* we create `cards`. Note that shuffle() shuffles list item *in place* using the fact that the list is mutable. That means you simply call the function and pass the list as an argument. The list gets modified, nothing is returned and nothing need to be assigned back of `filenames` variable.

Put your code into `code09.py`.

10.19 Let's have a break!

We covered a lot of ground, so it might be a good point to take a break and submit your code for my review.

10.20 Adding main game loop

At this point, we have a shuffle deck of cards that we show until a player presses a key. Modify the code to have the main presentation loop, similar to one we had when we experimented with PsychoPy stimuli. Previously, we used a logical `gameover` variable to control the while loop. Here, we will have two reasons to exit the loop: the player pressed an **escape** key or they won the game. Therefore, let us use a *string* `game_state` variable that is initialized to `"running"`. Repeat the loop while the `game_state` is equal to `"running"` but change it `"abort"` if a player pressed **escape**. You also need to replace waitKeys) with (getKeys.

Put your code into `code10.py`.

10.21   Detecting a mouse click

In the game, the player will click on individual cards to turn them over. 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 immediately below the line where you create the window itself.

Now, you can check whether the left button was pressed using mouse.getPressed() method. It returns a three-item tuple with `True`/`False` values indicating whether each of the three buttons are *currently being pressed*. Use it the main loop, so that if the player presses *left* button (its index in the returned list is 0), you change `"side"` of the first card (so, the card with index 0 in the list) to `"front"`. This assumes that you initialized card with their `"back"` shown, of course. If you run the code and click *anywhere*, this should flip the first card.

Put the mouse-click-processing code *before* drawing cards. At the moment, it makes no difference but will be useful later on, as it will allow us to draw the latest state of the card (i.e., right after it was flipped by a player).

Put your code into `code11.py`.

10.22   Position to index

Currently, the first card is flipped if you click *anywhere*. But the card you flip should be the card the player clicked on. For this we need to implement a function `index_from_position` that is an inverse of `position_from_index`. It should take an argument `pos`, which is a tuple of (`<x>`, `<y>`) values (a 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 for norm units) and by default the values you compute from them will also be float. However, an index *must* be integer, so you will need to wrap it in int() function call, before returning it.

Going backwards — from position to index — is (IMHO) easier. First, you need to think how you can convert an x coordinate (goes from -1 to 1) to a column index (goes from 0 to 3) given that you have 4 columns (draw a sketch on paper as it will make figuring out math simpler). Similarly, you translate y (from -1 to 1) into raw index given that there are only two rows. Once you know row and column index, you can compute the index itself, keeping in mind that there are four card in a row. As with `position_from_index`, I think it is easier to first play with formulas in a Jupyter Notebook, before turning the code into a function, documenting it, and putting it into `utilities.py`.

Put `index_from_position` into `utilities.py`.

10.23   Flip a selected card on click

Now that you have function that returns an index from position (don't forget to import it), you can flip the card that the player did clicked on. For this, you need to extend the card-flipping code inside the *if left-mouse button was pressed* code. Get the position of the mouse within the window by calling mouse.getPos(). This will return a pair 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. Change the "side" of a card with that index to "front". Test the code by turning different cards over, make sure that it is the card that you clicked on that gets turned. And a usual reminder, do not hesitate to put a breakpoint inside the if-statement to check the actual mouse position values and how they are translated into index, if things do not work.

Put your code into `code12.py`.

10.24   Keeping track of open cards

In the actual game, a player is allowed to flop only *two* cards at a time. If they match, they are removed. If not, they are flipped to their backs again. This means we need to keep track of which and how many cards are face up. We can always figure this out by doing a list comprehension scanning for cards that have their "side" as "face". But, mutable nature of dictionaries presents us with a simpler solution. We create a new list (let us call it `face_up`) and add cards to it. Mutable dictionary will not be copied but rather a reference to it will be present in both lists (same card dictionary will have two stickers on it, one from the `cards` list, one from `face_up` list). This way we know *which* cards are face up (those that are in the list) and we know how many (length of the `face_up` list).

However, you need to be careful not do add a card more than once (this will mess up our "how many cards are face up" number). There are two ways to do this. Assuming that `icard` is the index of the card, which you computed via `position_to_index()` from mouse position, you can simply check whether this card "side" is "front". Alternatively, you can check whether this card is already in the `face_up` list. Either way will tell you whether the card is face up. If it is not, you should set its "side" to "front" and add it to `face_up` list.

Implement this code, open a few cards. Then, use a breakpoint to pause the program and check that `face_up` list contains exactly these (this many) cards. If it has *more* then your face-up checks do not work. Put a breakpoint on them and step through the code to see what happens.

Put your code into `code13.py`.

10.25   Opening only two cards

Now we need to check whether a player open already two cards. In your code, mouse checks should be *before* the drawing code. This means that cards are drawn face up immediately after a click. Once they are drawn, check the length of `face_up`, if it equal to 2:

- pause the program for ~0.5 s[12] via wait, so that the player can see both cards.
- flip both cards back (i.e., set their `"side"` to `"back"`).
- remove them from `face_up` list (see .clear() method).

Put your code into `code14.py`.

10.26   Taking a matching pair off the table

Our code turns cards back even you found a matching pair but we need to take them off the table. Once you have two cards in the `face_up` list, you need to check whether they have the same chicken on them, i.e., their filenames are the same. If they are, you set `"show"` field to `False`. If not, you set their `"side"` to `"back"` (what your code is already doing). Either way, you still need to pause the program to allow the player to see them and to clear `face_up` list (they are either off the table or face down, definitely not face up).

We also need to modify our code to handle `"show"` field correctly. First, modify your drawing code to draw only the cards that should be shown. Second, when handling mouse click, you need to check both that the card is not face up and that it is shown (otherwise you can "open" invisible cards).

Put your code into `code15.py`.

10.27   Game over once all the cards are off the table

When your code works correctly, you can take all the card off the table, so that only the gray screen remains. However, that should be the point when the game finishes and congratulates you in your success. Write a function `remaining_cards` that will take the list with cards (i.e., our `cards` list) and will return how many cards are still shown (their `"show"` field is `True`). You definitely need a for for this but implementation can be very different. You could use an extra counter variable that you initialize to 0 and then increment by one

---

[12]Pick the timing you like!

(see += for a shortcut). Alternatively, you can use list comprehensions to filter out all cards that not shown and return the length of that list (a single line solution). Implement this function in *utilities.py* and use exit the loop by setting `game_state` to `"victory"`. After the loop, you can check the `game_state` variable and if the player was victorious, show a congratulatory message (TextStim, note that you don't even need to create a variable for it, you can create an object and call `.draw()` on it, i.e., `visual.TextStim(...).draw()`) and wait for a key press before you close the window.

Put your code into `code16.py`.

10.28 Do it fast!

There are different ways on how you can quantify speed in this game. You could look at the number of pairs the player had to open until clearing them up (the fewer, the better). Or, you could measure how fast the player did it in seconds. (Or use a combination of these two measures) Let us use the second option — total time taken — as an opportunity to learn about using PsychoPy clocks.

The two classes you will be primarily interested in are Clock and Countdown-Timer. The only difference between the two is that Clock starts at (and resets to) 0 and start counting *elapsed* time, so its getTime() method will return only *positive* values. In contrast, the CountdownTimer start with (and resets to) a value you initialized it with and starts counting *remaining* time down. Importantly, it will not stop once it reaches 0, so you will eventually end up with *negative* remaining time. Thus, for Clock you check whether the *elapsed* time is longer than some predefined value, whereas for CountdownTimer you start at a predefined value and check that the *remaining* time is above zero. Note it is not guaranteed that the remaining time will be exactly zero. If anything, it is extremely unlikely that this will ever happen, so never test for an exact equality with zero[13]!

Here, we are interested in the *elapsed* time, so Clock is the obvious choice. Create a clock before the game loop and use the elapsed time in the congratulatory message.

Put your code into `code17.py`.

---

[13]More generally, never compare float values to exact numbers. They are tricky, as the underlying representation does not guarantee that the computation will produce *exactly* the number that it should: `.1 + .1 + .1 == .3` is surprisingly `False`, try it yourself