Writing games with Python and PsychoPy

Alexander (Sasha) Pastukhov

2022-11-13

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.

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 in the missing information 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 (and without you realizing this as well). 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 (scikit-learn), deep 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).

1.6 About the material

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

---

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

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 2022.2.4 using Python 3.8 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.

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.

---

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

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 "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!

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

---

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

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 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!

---

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

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!

3.2 Reading the code

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].

My experience with programming in general and on this seminar in particular is that most problems you get stuck with are simple to the point of being dumb and obvious in retrospect[6]. Do not despair! It is not you, but just a consequence of

---

[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 than on the meaning and the story.

[6]Hindsight is always 20/20!

how wonderfully your brain is wired for pattern-recognition. Below are several suggestions that could help you to make reading code more robust.

3.2.1 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.

3.2.2 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[7]. 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.

3.2.3 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.

3.2.4 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 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.

---

[7]Recently, I spent half-an-hour trying to understand why two identical chunks of code give me different results. My son almost instantly found a difference (a missing comma in of the them) because for him it was just bunch of letters and numbers.

3.3 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, 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?

```python
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:

```python
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

```python
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`

```python
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

```python
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`.

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?

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

Do exercise #5.

---

[1]not really

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 to use print() function (we will talk about functions in general later). In the simplest case, you pass the value and it will be printed out.

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

or

```
print("five")
#> five
```

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

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

or

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

You can also pass more than one value/variable to the print function and all 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

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

What will be printed by the code below?

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

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

Do exercise #9.

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

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

You could, of course do `print(meal, dish, count)` but it would be nicer to
print "*I had* **10 pancakes** *for* **breakfast**", where items in bold would be the
inserted variables' values. For this, we need to use string formatting. Please
note that the string formatting is not specific to printing, you can create a new
string value via formatting and store it in a variable without printing it out or
print it out without storing it.

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 in 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:

```
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?

```
print('I had %d pancakes and either %d  or %d steaks for dinner'%(2))
print('I had %d pancakes and %d steaks for dinner'%(7, 10))
print('I had %d pancakes and %d steaks 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 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.
- Debugging code.
- Getting input from a user.
- Using comparison in conditional statements.
- Using indentation to group statements together.
- Using Python libraries.

---

[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

- 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 debugging 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 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

```
'''
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. 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.

---

[4]Or, at least, easier.

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.
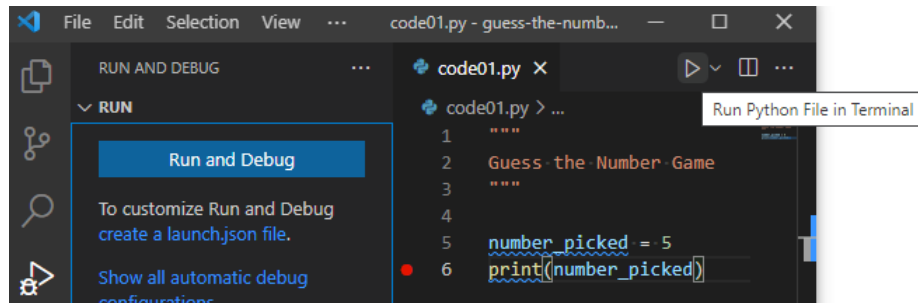
Figure 5.1: Running code without a debugger.

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).
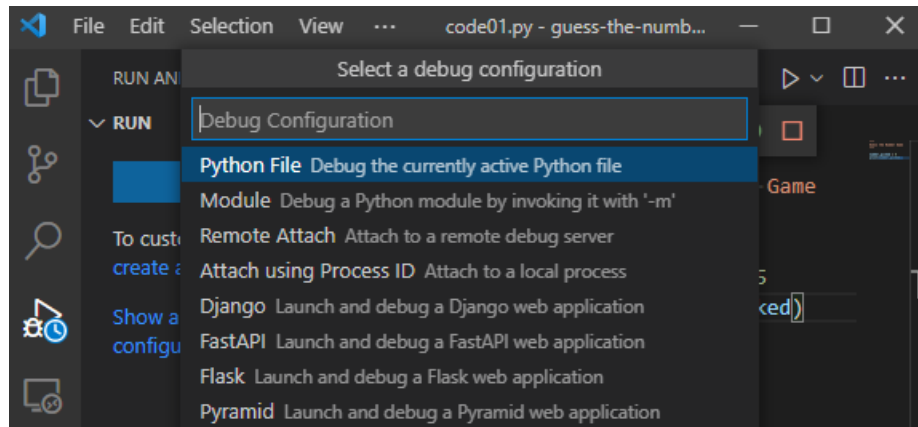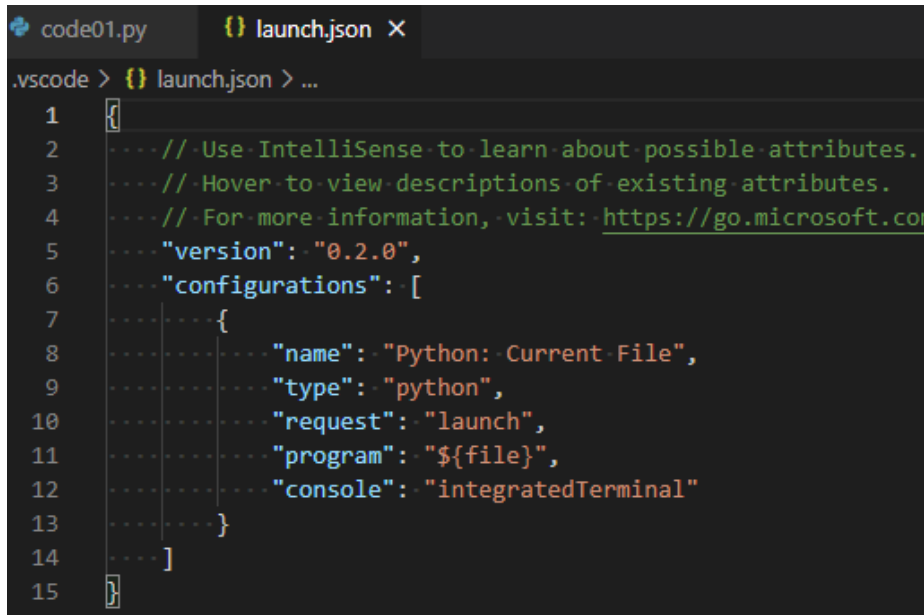
Figure 5.2: Selecting debugging configuration.

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:


```
code01.py        {} launch.json  ×
.vscode > {} launch.json > ...
  1   {
  2       // Use IntelliSense to learn about possible attributes.
  3       // Hover to view descriptions of existing attributes.
  4       // For more information, visit: https://go.microsoft.com
  5       "version": "0.2.0",
  6       "configurations": [
  7           {
  8               "name": "Python: Current File",
  9               "type": "python",
 10               "request": "launch",
 11               "program": "${file}",
 12               "console": "integratedTerminal"
 13           }
 14       ]
 15   }
```

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
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.

Figure 5.4: Active breakpoint.

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.

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

Figure 5.5: Program paused at the breakpoint.

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() *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 an 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 the has player entered (`guess`). We need to compare them to provide correct output message. For this, we will use conditional if statement:

```
if some_condition_is_true:
    # do something
elif some_other_condition_is_true:
    # do something else
```

```python
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).

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, in *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.

---

[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`.

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 or True` → `True`, `True or False` → `True`, `False or True` → `True`, but `False or 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

```
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):

---

[6]pun intended

```python
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.

```python
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)
```

```python
# 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.

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`.

---

[7]Did I already say never? Never!

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 anything, if all numbers are different.

Do not forget to document the new `code03.py` file and feel free 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 `1`. 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 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 as follows:

```python
# 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. Grab the exercise notebook before we start!

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 last 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 typically be a *verb* because function is about performing 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

---

[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.

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

---

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

[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.

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 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
    """
    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 a 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":
    ...
```

```python
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

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 functions 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)

# 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
```

---

[8]This is also not strictly true but you will have to wait until you learn about named parameters and default values

```
    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!).

```
# 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
  # happens to `x` in f1, stays in f1's scope.
```

---

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

```python
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. It should have no parameters (for now) and should return player's response. 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`.

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 our 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 based on player's response. 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?):

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

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

```python
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 a 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 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

---

[1] Also called properties

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

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

---

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

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 a 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.

---

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

```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 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).

```python
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.

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.

---

[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 is to draw first but in most cases it makes no difference.

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 unit systems!

8.8 Five unit 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[6].

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 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.

---

[6]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

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[7].

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[8]. 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. 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"`[9]. 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 you breakpoint inside the `if` statement, so that the program pauses only once you pressed a key (what happens if you put it on the `win.flip()` line?)

Put your code into *code10.py.*

---

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

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

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

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!

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.

Grab the exercise notebook before we start!

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 your 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

---

[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`, as `my_pretty_numbers[:]` returns a *different* list with *identical* content. The difference is subtle but important and we will return to it later when talking about mutable versus immutable types.

of the list[5]. For example, if you want to get the *last* element of the list, you can say `my_pretty_numbers[len(my_pretty_numbers)-1]` or just `my_pretty_numbers[-1]`. The last-but-one element would be `my_pretty_numbers[-2]`, etc. You can use negative indexes for slicing but keep in mind 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 `start:stop: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[6]. 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.

In Python lists are dynamic, so you can always add or remove elements to it,

---

[5]If you are coming from R, negative indexing is completely different in Python.

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

see the list of methods. 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[7]. However, All work and no play makes Jack a dull boy! So let us start with a basic PsychoPy scaffolding. Here the code structure:

```python
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).

---

[7]And we barely scratched the surface!

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:

```python
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[8]! Modify your code to use range() function in the for loop (how can you compute

---

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

the number of iterations from the length of the list?), use temporary variable as an index for the list to draw each item[9]. 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[10] 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.

Once you have the code, put a breakpoint and check that the value of the index variable matches what is shown on a screen[11].

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

---

[9]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.

[10]`itarget`? `imole`?

[11]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!

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):

```python
"""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

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`[12]. 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[13]. 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[14], 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.

---

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

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

[14]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).

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 and 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 in the Memory game.

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. Also, grab the exercise notebook!

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 contrast 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.

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

---

[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.

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 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

---

[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.

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 *the same list* to y, 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 behave the way you expect them to behave.

This variable-as-a-sticker, a.k.a. "passing value by reference", has very important implications for 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 mutable objects.

Do exercise #3.

10.4   Tuple: a frozen list

The wise people who created Python were acutely aware of the problem that the *variable-as-a-sticker* creates. Which is why, they added an **immutable** version of a list, called a tuple. It is a "frozen" list of values, which you can loop over, access its items by index, or figure out how many items it has, but you *cannot modify it.* No appending, removing, replacing values, etc. For you this means that a 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.

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

You can loop over it, e.g.,

```python
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)

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

# throws AttributeError: 'tuple' object has no attribute 'append'
i_am_a_tuple.append(4)
#> Error in py_call_impl(callable, dots$args, dots$keywords): AttributeError: 'tuple'
```

Same goes for trying to change it

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

# throws TypeError: 'tuple' object does not support item assignment
i_am_a_tuple[1] = 1
#> Error in py_call_impl(callable, dots$args, dots$keywords): TypeError: 'tuple' objec
```

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

---

[6]Well, at least me!

link to the original list object is now broken. You can turn a list into a tuple using `tuple()`. Keeping in mind that `tuple()` creates a frozen copy of the list, what will happen below?

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

Do exercise #4.

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

Conversely, you "unfreeze" a tuple by turning it into a list via `list()`. Please note that it creates **a new list**, which has no relation to any other existing list, even if values 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 (abstract 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 much 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.

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
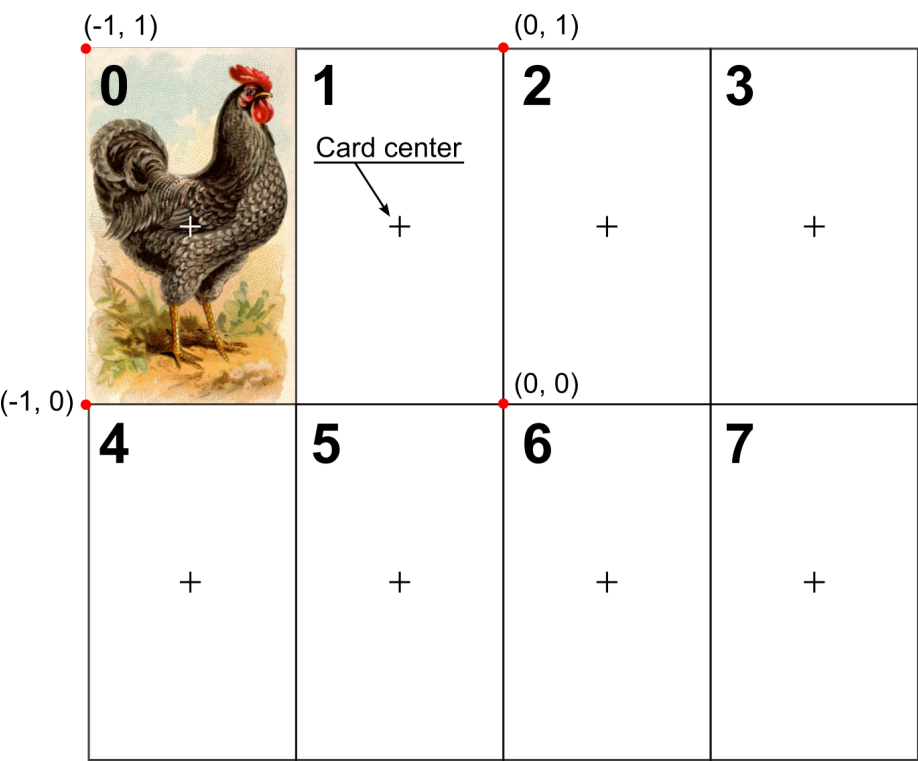
Figure 10.1: Card location index

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 Note-
book, 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. Remem-
ber, 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 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 back (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 these values 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 to `"back"` because, initially, all cards are face down. However, you can always set it temporarily to `"front"` to see how the cards are distributed.

---

[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.

   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`.

Create a dictionary variable (name it `card`) and fill it with 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 your
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 to 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 *Im-
ages* folder. This is covered by os library that contains various utilities for
working with your operating system and, in particular, with files and directo-
ries. 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")`, assum-
ing that *Images* is a subfolder in your current directory - or an absolute path
`os.listdir("E:/Teaching/Python/MemoryGame/Images")` (in my case)[8].

---

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

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* 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.

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

---

[9]A very arbitrary example!

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

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

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

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

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

Do exercise #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.

10.15   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]

---

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

- 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.

10.16 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 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 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
```

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

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

10.17 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`.

10.18   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`.

10.19   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`.

10.20 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.

10.21 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`.

10.22 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`.

10.23 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 row 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`.

10.24   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 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`.

10.25   Keeping track of open cards

In the actual game, a player is allowed to flip 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`.

10.26   Opening only two cards

Now we need to check whether a player opened exactly 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[11] 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`.

10.27   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

---

[11]Pick the timing you like!

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`.

10.28   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 on 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 (see += for a shortcut). Alternatively, you can use list comprehensions to filter out all cards that are 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`.

10.29   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[12]!

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`.

---

[12]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

Chapter 11

Christmas special

Today we are going to program a Christmas-special. However, this is still an opportunity to learn something new. You will learn about zipping lists and we will start offloading settings into a separate file. Here is how the Christmas tree looks for me:

11.1 Chapter concepts

- Building Christmas spirit
- Zipping over lists
- Loading setting from JSON or YAML file.

11.2 Christmas tree

Let us start our Christmas decoration with a Christmas tree. You can download the one I've found (created by isaiah658) or find an image that you like. Create your basic PsychoPy code to create a window (we will be using Circle later, so think about suitable units), an ImageStim with a tree, draw it and wait for any key press.

Put your code into `code01.py`.

11.3 Christmas tree decoration

For the decoration, let us use Circle objects of various sizes and color. We could create each one separately with its own custom hard-coded values, but let us instead create three constants that are lists of equal length that describe, respectively position of each ball (`BALL_POS` would be a good name, each entry should be a tuple of `(x, y)`), size (`BALL_SIZE`), and color (`BALL_COLOR`, stick

109

to `"red"`, `"blue"`, and `"yellow"`, this limited selection of specific colors will be important later when we animate them).

Create a list of balls by iterating over these three lists. You have two choices, you can either use an index variable, building an index via range() using len() of one of the lists (they should all be of the same length). But let use a cool trick of iterating over a zip() of lists. zip() gives you a tuple combining one element from each list that you can unpack on the fly as in the example below (note that loop variables will receive values in the order that you used for lists).

```python
numbers = [1, 2, 3]
letters = ["A", "B", "C"]
for  a_number, a_letter in  zip(numbers, letters):
  print("%d: %s"%(a_number, a_letter))
#> 1: A
#> 2: B
#> 3: C
```

You can zip as many lists as you want. We, obviously, want three. Decide on whether you want to create `balls` as an empty list and then append each newly created Circle to it in the loop or use list comprehension. Do not forget to draw the balls and think about what you should draw first: the tree or the balls. Experiment with position and sizes to makes it look just perfect.

Put your code into `code02.py`.

11.4   Twinkle, twinkle, little star

Now let us make our Christmas balls twinkle, as in the video. The idea is that only one color is "active" at a time. The balls of that color are "on" and balls of other color are "off" (white or gray, or some other color of your liking). Now our display becomes dynamic, so you need to have a game loop and with an opportunity to exit the program by pressing *escape*.

For this, we need to define a list of colors ("red", "blue", "yellow") that we can cycle through and an variable that hold the index of the currently active color (I've called it `icolor`). Every X seconds (I do it every 0.5 seconds, define this as a constant, e.g., `TWINKLE_DURATION`), increment this index, so that the next color in the list becomes active. Note that you have an out-of-range problem: When you initialize `icolor` to 0 and increment it by 1 three times, your index is already too large (3, the length of our `colors` list is 3, so the maximal index is 2). You can either use an `if` to check for that or you can use a remainder operator `%` (think about the remainder if you divide *any* positive value by the length of the `colors` list).

Once you need to update whether balls are "on" or "off", you need to loop both through the balls and their colors in the original `BALLS_COLOR` list (when you use a string with color name, it gets translated into an RGB value, so we

cannot compare it directly). Again, you can use zip() to loop simultaneously through Christmas balls and their color. If their color matches the active one, their `fillColor` should be that color. If not, their `fillColor` should be some "neutral" / "off" color (white? gray?).

To keep track of time, you will need a timer variable, use either Clock or CountdownTimer. Once the `TWINKLE_DURATION` elapsed, update the active color, all the balls, and do not forget to reset the timer.

Put your code into `code03.py`.

11.5   Let's make some noise!

Let add some Christmas music! Download Deck the Halls version by Kevin MacLeod[1]. For this, we will use sound module of PsychoPy library that generate sounds on the fly and also play audio files in various format such as wav or ogg (but not mp3!). Unfortunately, sound it surprisingly tricky, there are many libraries that might be used by PsychoPy (as of end of 2021 PsychoPy lists four backends that it might use), and things sometimes break. Thus, if the music does not play for you, ask me and we will try to set your sound libraries up.

Using sound is very simple. First, you need to import the `Sound` class as suggested in the manual:

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

Then, you need a new object of `Sound` class supplying the file name as the first parameter (I called the variable `song`). Right before start the loop, you `.play()` the sound. Note, if you want to play the same sound again, you need to "rewind" it by explicitly calling its `.stop()` method (for some reason, the sound stops at the end but does not gets "rewind", so when you try to play it again and notices that it is already at the end and stops without playing anything).

Put your code into `code04.py`.

11.6   Settings file formats

So far, we either hard-coded specific values or defined them as constants (a better of these two approaches). However, this means that if you want to run your game with different settings, you need to modify the program itself. And if you want to have two versions of the game (two experimental conditions), you would need to have two programs with all the problems of maintaining virtually identical code in several places at once.

---

[1]Deck the Halls B by Kevin MacLeod http://incompetech.com Creative Commons — Attribution 4.0 International — CC BY 4.0 Free Download / Stream: https://bit.ly/deck-the-halls-b Music promoted by Audio Library https://youtu.be/RzjZ-WdVeyk

A better approach is to have separate files with settings, so you can keep the code constant and alter specific parameters by specifying which settings file the program should use. This is helpful even you plan to have a single set of setting as it separates code from constants, puts the latter all in one place and makes it easier to edit and check them. There are multiple formats for settings files: XML, INI, JSON, YAML, etc. Our format of the choice for today will be JSON. However, this is a question of taste. Personally, I like YAML for subjective reasons (fewer curly brackets and quotation marks), but you are free to use any format you like. As you will see, this makes little difference for the actual Python code.

11.6.1   XML

XML — an Extensible Markup Language — looks similar to HTML (HyperText Markup Language). Experiments designed using PsychoPy Builder interface are stored using XML files but with .psyexp extension. A settings file for our Christmas programin XML could look like this

```xml
<Balls>
  <Ball>
    <Position>
      <x>0.1</x>
      <y>0.2</y>
    </Position>
    <Size>0.01</Size>
    <Color>red</Color>
  </Ball>
  <Ball>
    <Position>
      <x>0.2</x>
      <y>0.1</y>
    </Position>
    <Size>0.02</Size>
    <Color>yellow</Color>
  </Ball>
  ...
</Balls>
<Timing>
  <Twinkle duration>0.5</Twinkle duration>
</Timing>
```

The advantage of XML is that it is very flexible yet structured and you can use native Python interface to work with it. However, XML is not easy for humans to read, it is overpowered for our purposes of having a simple set of unique constants and its power means that using it is fairly cumbersome (I use \ to split a single line into many lines).

```python
from xml.dom import minidom
settings = minidom.parse('settings.xml')
# this will give you string "0.4"
size = settings.getElementsByTagName("Balls")[0]. \
                getElementsByTagName("Ball")[0]. \
                getElementsByTagName("Size")[0].firstChild.data
```

11.6.2 INI

This is a format with a structure similar to that found in MS Windows INI files.

```ini
[Balls]
    x = 0.1, 0.2
    y = 0.2, 0.1
    size = 0.01, 0.02
    color = red, yellow
[Timing]
    TwinkleDuration = 0.5
```

As you can see it is easier to read and Python has a special configparser library to work with them. The object you get is, effectively, a dictionary with additional methods and attributes. However `ConfigParser` does not try to guess the type of data, so all values are stored as *strings* and it is your job to convert them to whatever type you need, e.g., integer, list, etc.

```python
import configparser
settings = configparser.ConfigParser()
settings.read('settings.ini')
settings['Balls']['size'] # this will give you a string '0.01, 0.02'
```

11.6.3 JSON

JSON (JavaScript Object Notation) is a popular format to serialize data for web applications that use it to exchange data between a server and a client.

```json
{
  "Balls": {
    "position": [[0.1, 0.2], [0.2, 0.1]],
    "size": [0.01, 0.02],
    "color": ["red", "yellow"]
  },
  "Timing": {
    "Twinkle duration" : 0.5
  }
}
```

You can parse any *string* in JSON format into a dictionary in Python using json

module. Its advantage over INI files is that JSON explicitly specifies data type (i.e., strings are in quotation marks), so it converts it automatically. Note that unlike configparse, json module does not work with files directly, so you need to open it manually (ignore the `with` magic for a moment, you will learn about it in detail when we will talk about context managers).

```python
import json
with open('settings.json') as json_file:
    settings = json.load(json_file)


settings["Balls"]["size"] # this will give a list [0.01, 0.02]
```

11.6.4   YAML

YAML (YAML Ain't Markup Language, rhymes with camel) is very similar to JSON but its config files are more human-readable. It has fewer special symbols and curly brackets but, as in Python, you must watch the indentations as they determine the hierarchy.

```yaml
Balls:
  position: [[0.1, 0.2], [0.2, 0.1]]
  size: [0.01, 0.02]
  color: ["red", "yellow"]
Timing:
  Twinkle duration : 0.5
```

You will need to install a third-party library pyyaml to work with YAML files. You get the same dictionary as for the JSON

```python
import yaml
with open("settings.yaml") as yaml_stream:
    settings = yaml.load(yaml_stream)


settings["Balls"]["size"] # this will give a list [0.01, 0.02]
```

11.7   Using settings

Look at your *code04.py* and identify constants and hard-coded values that you should put into a settings file. E.g., definitely constants that describes Christmas balls and twinkle duration but, possibly, also the size of the window, name of the Christmas tree and song files, etc. In general, I put every such value into settings even if it used only once (as with the size of the window) because then I know that *all* constants are the settings file. This way there is a single, nicely organized place to check and I do not need to search through the code to figure a specific value out.

Once you transferred all your constants into the settings file (use either JSON

or YAML), add the code that loads it at the very beginning and use settings dictionary in place of constants.

Put your code into `code05.py`.

11.8 Merry Christmas and a Happy New Year!

Chapter 12

Flappy Bird

Today we will stat developing a *Flappy Bird* game. You control a bird that must fly through the openings in the obstacles but your only action is to "flap the wings" in order to counteract the effect of gravity. This is how the game will look like at the end.

12.1   Chapter concepts

- Object-oriented programming
- `@property`

We will use this game as an opportunity to learn more about object-oriented programming. You already know how to use classes, now you get to create them and see how it makes your life easier.

12.2   Object-oriented programming

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

12.2.1   Classes and objects (instances of classes)

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

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

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

12.2.2   Encapsulation

Putting all the data (properties) and behavior (methods) inside the class simplifies programming by ensuring that all relevant information can be found in its definition. Thus, you have a single place that should hold *everything* that defines object's behavior. Contrast this with our approach in previous seminars where cards as dictionaries were separate from functions that created them. Today, you will see how encapsulating everything into classes turns this mess into a simpler and easier-to-understand code.

12.2.3   Inheritance / Generalization

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

Using the Homo sapiens example from above. Humans, chimpanzees and gorillas are all different species but we share a common ancestor. Hence, we are different in many respect, yet, you could think about all of us as "apes" that have common properties such as binocular trichromatic vision. In other words, if you are interested in color vision, you do not care what specific species you are looking it, as all apes are (roughly) the same in that respect. Or, you can move

further down the evolution tree and think about us as "mammals" that, again, have common properties and behavior, such as thermoregulation and lactation. Again, if you are interested *only* in whether an animal has thermoregulation, knowing that it is a mammal is enough.

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

Where is other way of achieving common behavior (generalization) in Python without orderly inheritance, such as "duck typing"[2] but it will be a topic of another seminar.

12.2.4 Polymorphism

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

12.2.5 A minimal class example

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

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

---

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

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

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

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

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

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

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

---

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

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

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

12.2.6 `add` method

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

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

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

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

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

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

12.2.7 **Flexible accumulator with a `subtract` method**

Now lets us create a new class that is a *descendant* of the `Accumulator`. We will call it `FlexibleAccumulator` as it will allow you to also *subtract* from the

---

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

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

total count. You specify ancestors (could be more than one!) in round brackets
after the class name

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

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

Add `subtract` method to the `FlexibleAccumulator` class in a Jupiter note-
book. Add testing.

12.3   Method arguments

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

Create `SuperFlexibleAccumulator` class and definesuper flexible `add` and
`subtract` methodsthat have `value` parameter (in a Jupiter notebook).Test
them!

12.3.1   Constructor arguments

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

Modify the code so that you pass the initial value that total is set to, instead of
zero.

Add `initial_value` parameter to the constructor of the `SuperFlexibleAccumulator`
class in a Jupiter notebook. Test it!

12.3.2   Calling methods from other methods

You can call a function or object's method at any point of time, so, logically,
you can use methods inside methods. Let's modify our code, realizing that

subtracting a value is like *adding a negative* value. Modify your code, so that `.subtract()` only negates the value before passing is to `.add()` for actual processing. Thus, `total` is modified *only* inside the `add()` method.

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

12.3.3 Local variables

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

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


counter = Accumulator(1)
```

What about in this case?

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


counter = Accumulator(1)
```

12.4   Flappy Bird: the humble beginnings

We will start with a basic scaffolding for our program. Download the bird image[7] and put it into a folder where you will store the code. Create a basic code that uses settings file that defines minimal setting for a window (size) and a bird (image file). Organize it hierarchically, as follows, as this will help us keep settings for different classes organized.

```json
{
  "Bird": {
    "Image" : "Blue-Bird.png"
  },
  "Window": {
```

---

[7]Created by Madison Kingsford.

```
    "Size": [800, 600],
  }
}
```

Create a window using this specified size and an ImageStim using the filename from the settings file. Add a basic game loop in which you repeatedly draw the bird (should appear right at the center of the screen) and check for a key press (*escape* should exit the game).

Put your code into `code01.py`.

12.5 Flappy Bird class

Our flappy bird is, effectively, an image but we would like it to have additional behaviors, like, automatically falling down due to gravity, flying up due to flapping its wings, etc. There are several way we can do this. We can keep the image in ImageStim and write additional functions to handle it (the way we did previously). We could create an new class `FlappyBird` that will have the ImageStim as its attribute. Or, we could utilize the power of inheritance and build `FlappyBird` class on top of the ImageStim. This means less work for us, so that is the path we will follow.

Create a new file that will contain your `FlappyBird` class. Here is how it should look like:

```
"""Your comment on what this file contains.
"""
# import libraries, which ones do you need?

class FlappyBird(visual.ImageStim):
  """
  FlappyBird class based on ImageStim
  """
  def __init__(self, win, settings):
    """
    Constructor.
    """
    super().__init__(win, image=settings["Image"])
```

In the code above, I defined `FlappyBird` as a descendant of the ImageStim. To make the latter work, we need to initialize it properly by calling it constructor. This is what `super().__init__(...)` call does: Calls constructor of the ancestor class to enable all the magic that we want to reuse. Recall that ImageStim needs at least two parameters: a PsychoPy window that the stimulus will belong to and an image (a filename in this case). Here, I assume that when I create a bird object (call the constructor), I pass *two* parameters (again, `self` comes "for free", so you do not pass it explicitly but assume that it is the *first* argument

that you get): the [window]((https://psychopy.org/api/visual/window.html# psychopy.visual.Window) that we created plus a dictionary with settings for the bird (there will be more settings, so it would be practical to pass the whole dictionary instead of passing one parameter at a time).

Copy paste that code (plus add appropriate imports and comments) and use `FlappyBird` class instead of ImageStim. Note that `FlappyBird` inherits *all* its functionality from ImageStim, so, apart from how you create it, you can use it in exactly the same way. Meaning, you do not need to modify anything else in your code (told you, it would save us time and effort!).

Put `FlappyBird` class code into a separate file. Use it instead of `ImageStim` in `code02.py`.

12.6 A properly-sized bird

Our bird is very cute but is way too large. Add a new setting for it (I suggest calling is `Size` and setting it to 0.1) and then use it inside the constructor adding `size=...` to `super().__init__` call. Do you need to change anything in the main code?

Add bird size setting. Use it in `FlappyBird` class constructor.

12.7 Flappy Bird is falling down (my dear lady)

Before our bird flies, it needs to learn how to fall down. Falling down is just a change of bird's vertical position based on bird's vertical speed. We already have a property for the (horizontal and) vertical position: `self.pos`, a tuple with `(x, y)` position of the center of the image. But we do need an additional new attribute that would encode bird's vertical speed. Create it in the constructor (if you forgot how to do it, take a look above on how we create the `total` attribute for `Accumulator` class) and call it `vspeed`. Also, create a new setting (I would call it `"Initial vertical speed"`) and set it to `-0.01`, use this setting in the constructor to initialize `vspeed`.

Now we also need a method that would update bird's position based on its (current) speed. Create this method below the constructor (does it need any parameters beyond compulsory `self`?). It should simply compute $y_{new} = y + vspeed$ and assign y_{new} back to `pos` attribute (note that you cannot assign only y coordinate, you have to pass the tuple `(x, y)` reusing original `x` value from `pos`). Do not forget to document the new method!

Now you need to call the `update()` on each frame before drawing the bird. This should make your bird fall of the screen! (Experiment with `"Initial vertical speed"` setting to make it fall faster or slower or even upwards!)

Update `FlappyBird` class. Use `update` method in `code03.py`.

12.8   Timing the fall

Currently, the speed of our bird's fall is measured in `norm units / frame`. This works but these are not the most convenient units to think in. Plus, it relies on PsychoPy (and the rest of our code) to ensure that time between individual frames is exactly the same. This is *mostly* the case and is an occasionally slow bird is not a big problem for a game. However, that might be a problem for an actual experiment that requires precise timing of movement. Thus, we need to think about vertical speed in units of `norm units / second` and measure time between calls of `update()` method ourselves.

Create a new Clock attribute that will count the time elapsed since it last reset (I would call it `frame_timer`). Modify the `update()` method to compute $y_{new} = y + vspeed * T_{elapsed}$, where $T_{elapsed}$ is the time elapsed between frames. Do not forget to reset the timer! (What will happen if you do forget?)

Now set your `"Initial vertical speed"` to some reasonable value (e.g., 0.5) and check that the time it takes for the bird to fall of the screen looks reasonable (for 0.5 norm units / second it should be off the screen in two seconds).

Update `FlappyBird` class with a timer.

12.9   It is all Newton's fault

Now let us add gravity, so that the speed of falling would be constantly changing. Create a new setting and call it `"Gravity"`. Set it to `-0.5` (units are `norm units per second squared`) but experiment with different values later on. Acceleration due to gravity changes vertical speed just like speed itself changes the vertical position. Update your `update` method to change the speed based on acceleration given the elapsed time. What do you need to update first, the speed or the location? Also, think about how you will store the acceleration: It is in settings parameter that exists only in the constructor. You can either store it in a new attribute or store all settings in an attribute for later use.

Update `FlappyBird` class with acceleration due to gravity.

12.10   Flap bird, flap!

Let us add ability of the bird to "flap" in order to stay in the air. First, create a new setting `Flap speed` and set it to `0.4` (as usual, feel free to experiment!). Add a new method `.flap(self)` and inside simply set `vspeed` to `Flap speed`. Thus, a single flap sets the bird flying up with `Flap speed` speed which, however, will be constantly reduced by the accelartion due to `Gravity` so the bird will eventually start falling down again.

In the main code, check for `"escape"` and `"space"` keys. If the latter is pressed, call `.flap()` method of the bird. Check that you can keep the on the screen by

timing the space button presses or can make it fly upwards off the screen.

Add `flap` method to `FlappyBird` class. Use it in `code04.py` whenever player presses *space*.

12.11   Stay off the ground

In our game, the player can lose either if they hit an obstacle (we do not have any yet) or if the bird drops below the ground level, i.e., the bottom edge of the window. Create a new method `is_airborne()` that will return `True` if `y` position of the bird is above `-1` (note, you do not need an explicit `if` for this, nor do you need to write `True` or `False` anywhere, think how this can be done without these).

In the main loop add the check for `bird.is_airborne()` condition so that it continues until player presses `"escape"` or the bird hits the ground.

Add `is_airborne` method to `FlappyBird` class. Use it in `code05.py` as an additional conditionfor the game loop.

12.12   Computed attribute `@property`

As was explained the in "Object-oriented programming" section above, properties describe state of an object, whereas methods describe what an object can do or what you do to an object. Our `is_airborne()` method breaks this logic: It describes the *state* of the bird but we *call* it (use it) as a method. What we have here is a *computed* property that is inferred from other properties of an object. In our case, we infer property `is_airborne` from `y`. We could, of course, make `is_airborne` into a real property by defining it in the constructor and then updating it inside `update()` method. However, we will instead use a cool feature (called decorators) to turn a method into a read-only property. The only thing you need to do is to add `@property` decorator right above the `def is_airborne(self):` line and drop brackets when using it in the main loop (so just `bird.is_airborne` instead of `bird.is_airborne()`).

`@property` tells Python that the method right below **will** (must!) return a value and that outside world should see it not as a method but as a *property*. You can use it to make properties read-only, so that they could not be (easily) changed from outside or to create properties that are computed on-the-fly as in our example.

Note that difference is not so much of practical implementation (changes we made to the code were minimal) but of a conceptual nature: Object's states should be properties not methods. In our small example this may look like an overkill but in a moderately complex project even small conceptual blurring of lines could make it harder to understand the code.

Turn `is_airborne` into a property. Use it as property in `code06.py`.

12.13   An opening in an obstacle

We aim of the game is for the bird to fly avoiding obstacles. An obstacle consist of two rectangle, one that protrudes from above and another one from below. The opening between them gives the bird an opportunity to fly through. So let us begin by writing a code (in the Jupyter notebook) that would generate a random opening, characterized by `y_bottom` and `y_top` based on four parameters:

- `lower_margin` : the lower margin relative to the bottom of the screen, i.e., the `y_bottom` cannot be closer to -1 than that.
- `upper_margin` : the upper margin relative to the top of the screen, i.e., the `y_top` cannot be closer to 1 than that.
- `min_size` : the minimal size of the opening, i.e., minimal distance between `y_top` and `y_bottom`.
- `max_size` : the maximal size of the opening, i.e., maximal distance between `y_top` and `y_bottom`.

Write a code that assume certain values for each parameter (e.g., `lower_margin = 0.2`, `upper_margin = 0.2`, `min_size = 0.2`, `max_size = 0.4`) and generates a random pair (`y_bottom, y_top`) that satisfies the conditions.

Write a random opening code in a Jupiter notebook.

12.14   An obstacle

Now let us create an `Obstacle` class (put it into a separate file). It will consist of two rectangles, on protruding from the top and the other one from the bottom, with a random opening in between. At the moment you need six settings to describe an `Obstacle`: The four parameters that define a random opening plus a width of rectangles and their color. Describe them as a separate group in the settings files (probably under `"Obstacles"`) and pass them to the constructor of the `Obstacle` class.

In the constructor, generate a random opening (you have the code for this already) and create the two rectangles both `width` wide, one going from the top till `y_top`, another one from the bottom till `y_bottom`. Place both horizontally at the right edge of the window but so you could see them. Decide on how you will store the two rectangles, you can put them into two different attributes (e.g., `upper_rect` and `lower_rect`) or have them in a list. I would suggest the latter approach, as it will simplify your future code. Think about which parameters you need for the `__init__()` constructor function.

You also will need a `draw()` method that simply draws both rectangles. Implement the class in a separate file, then create and draw a single obstacle in the

main code to check that it looks right.

Create `Obstacle` class in a separate file. Use it in `code07.py`.

12.15   A moving obstacle

Conceptually, our bird fly towards an obstacle but instead we will induce perception of its motion by moving obstacles from right to left. Define a new `speed` setting for an `Obstacle` class, it should be in `norm units per second` and create an `update` method that would change horizontal position of both rectangles based on the amount of time that passed between the calls. This is similar to how we updated bird's position based on its speed, so follow the same logic and think about which additional attributes you require and how you store and use any relevant information.

Call `update()` at the same place that you update bird's location in the main loop and check that the obstacle is moving from right to left.

Add `update` method to `Obstacle` class. Use it in `code08.py`.

12.16   Hitting the wall

At the moment, our bird is flying through the obstacle like it is not there. But it is! Luckily for us, PsychoPy makes implementing this very easy, as it can check whether two stimuli overlap using overlaps() method of one of them (and the second stimulus is passed as an argument).

Thus, to check whether the bird hit the wall, we just need to create a method (let us call it `check_if_hit`) in an `Obstacle` class that will take a bird object and check whether it overlaps() with one of the rectangles. Remember, our `FlappyBird` is a descendant of the ImageStim, so we can pass it to the overlaps() method directly (advantage of using the inheritance!)

In the main game loop, add the check for the bird *not* hitting the obstacle to the main condition (so now you should have three things to check for). Test your code by flying the bird into the wall. Also, by flying your bird through the opening. Note that if our settings make it too tricky, modify them to make the opening larger.

Add `check_if_hit` method to `Obstacle` class. Use it in `code09.py`.

12.17   An obstacle manager

A game with a single obstacle is no fun but before we add more, we need a class that will manage them for us. Let us call it `ObstaclesManager`. For the moment, it will simply wrap up all functionality that we implemented in the main script. In the constructor, it should create a *list* attribute for obstacle

object and add a first one, it should also implement methods `draw()`, `update()`, and `check_if_hit()` that draws, update, and checks for an overlap with a bird for *all* obstacles on the list. For the moment we will still have just one of them in the list but implementing things in the loop means it will be easier to add more. Create the class and then use it in the main script.

There should be no actual changes how the game plays, just the code refactoring. However, it helps us to hide the management part from the main script (making it easier to understand) and, if you did everything correctly, the code should "just work" once you replace `Obstacle` with `ObstaclesManager` object.

Create `ObstaclesManager` class. Use it in `code10.py`.

12.18   Loads of obstacles

Now we are ready to add more obstacles. You need to update the `update` method of the `ObstaclesManager` so that add a new obstacle to the list after a random time interval. Define a new settings `Spawn time`, a list of two values that define a minimal and maximal time between spawns, and create a CountdownTimer (or a Clock, remember they differ only in whether time is subtracted or added) attributed. Whenever the timer is up, add a new obstacle to the list, generate a new delay, and set the timer again. Note that now you need settings and a Window in the `update` because they are required for creating a new obstacle. Think how you can store them for later use in the constructor.

Do you need to change anything in the main script? Check that more obstacles appear over time!

Update `update` of the `ObstaclesManager` class.

12.19   Removing redundant obstacles

Once the obstacle is past the left edge of the screen (its x-axis position is less than `-1`) we need to remove it from the list. Otherwise, we will waste a lot of time and memory keeping track and drawing obstacles that are neither relevant, nor visible. Think about how you would implement this before reading further.

First, we need to compute the horizontal position of an obstacle. You can deduce it from `pos` attribute of one of the rectangles and implement this use of attribute (`pos` of `Obstacle`) of an attribute (`obstacles` of `ObstaclesManager`) directly. However, working with attributes of attributes makes code harder to read and to maintain. Instead, add a new computed property `x` to `Obstacle` class that would return a single number (horizontal position) using `@property` decorator we used for the bird's `is_airborne` dynamic attribute.

Update the `update` method of the `ObstaclesManager` to check the position the *first* obstacle in the list. If it less than -1, simply pop it from the list. Why the

first one? Because any other obstacle in the list was added later and, therefore, must be further to the right. Why *only* the first one? If we assume both a reasonable motion speed and a reasonable spawn delay, it is very unlikely that more than one obstacles reaches the left edge at the same time.

Note that for safety reasons, you must first check that the `obstacles` attribute is not empty! Hint, when used in a conditional statement directly, empty list evaluates to `False`. Debug the code to make sure that obstacles are indeed removed. You can either use a different cut-off point (e.g., -0.25) to see that easier or put a break point at the line that pops the redundant obstacle (better still, do both!)

Update `update` of the `ObstaclesManager` class.

12.20   Keeping the score

It is hard to brag about your bird-flying skills, if you do not how many obstacles did you fly through. Let us add the score! First, create a TextStim (call it `score_text`) and put it somewhere one the screen, e.g., a top-left or top-right corner. Initialize its text to `"0"`. Draw it in the main loop. Make sure it works before you continue.

To keep the score, we need to count the number of obstacles that the bird clears on every frame and add it to the total score. As with the removal of redundant obstacles, it will be either zero or one obstacle that crossed the middle of the screen. But in this case, it is not the leftmost that we have to check but the *leftmost* among those that did not yet cross the mid-line. There are different way you can approach this, so think how you would approach this before reading on. And if you came up with a different plan, by all means – implement it!

My way of doing this is to introduce a new attribute `scored = False` and a method `score()` to the `Obstacle` class. In the `score()` method, if the object crossed the `0` line *and* has not been scored, it marked as `scored` and the method returns `1`. Otherwise, the object was either already scored or did not cross the mid-line yet, so it returns `0`. Next, I added a `score()` method to the `ObstaclesManager` that simply computes the total score (sum of) scores of all obstacles in the list. In the main script this score is added to a `score` variable that, in turn, is used to update `score_text`.

Update your code anduse it in `code11.py`.

12.21   A foundation

This is just a foundation of a game, so feel free to add to it. Animated bird? Difficulty levels? Different kinds of obstacles? High-score table?

Chapter 13

Guitar Hero: staircase and iterator functions

13.1 Getting the difficulty just right

In game design, one of the hardest things to get right is difficulty. Make your game too easy and it will be boring. Make it too hard and only hardcore fans will play and only for an achievement. Thus, you would like to make your game hard enough to push a player to the limit but not much harder than that, so not to frustrate them. One way to solve this conundrum is to create different preset difficulty levels. An alternative way is to make a game that adapts its difficulty to the player.

The same is true for psychophysical experiments. You want to test ability of your participants to perform a certain task at their limit for one simple reason: At this *threshold* point influence of any additional factor, whether positive or negative, is most pronounced. For example, use an unusual stimulus configuration or increase attentional load and performance will probably drop. Allow to preallocate attention via cuing or use a prime that is congruent with a target and performance is likely to improve. Of course, these manipulations will have the same overall effect also when the task is particularly easy or maddeningly hard but it will much more difficult to *measure* this effect. It is one thing if performance drops from 75% to 65% than if it goes from 98% to 95% or from 53% to 52%[1] or vice versa. The silliest thing you can do is to *hope* that performance will allow you too see the effect of the factors that you manipulated. In things like these, knowledge and careful design is definitely superior to hope.

Thus, you want performance of your participants to be approximately in the middle between the ceiling (100% performance, fastest response times, super

---

[1]Here, I assume that 50% is chance level performance.

easy) and the floor (chance level performance, slowest response times, super hard or even impossible). But how do you know where this magic point for a *particular* person is? Particularly, if the task is novel so you have little information to guide you[2]. The solution is to adjust the difficulty on-the-fly based on participant's responses. For example, if you have a two-alternatives-forced-choice task, you can use a two-up-one-down staircase (difficulty increases after two correct responses and decreases after one mistake) that targets 70.7% performance threshold. There are different methods and even different ways to use the same core method (e.g., does the step stays constant or changes, what is the run termination criteria, etc.), so it is always a good idea to refresh your memory and read about adaptive procedures when designing your next experiment.

In our game, we will use a very simple 3-up-1-down staircase: get the three responses correct on a row and things get faster, make a mistake and the game slows down. We'll see how fast you can go! First, you will implement it by hand and then we will use its PsychoPy implementation.

13.2 Guitar Hero

Today, we will program Guitar Hero game. In the original game, you must play notes on a guitar-shaped controller pressing buttons at the right time, just like when you actually play music on a guitar. On the one hand, it is a straightforward and repetitive motor task. On the other hand, take a fast and complicated music piece and it'll take many minutes or even hours of practice to get it right. It is a lot of fun, as music cues and primes your responses. The same idea of music-synchronized-actions was used in *Raymon Legends* music levels where jumps and hits are timed to drums or bass. It is a bizarrely cool dance-like sequence and a very satisfying experience, also when watching pros to do it (I happened to have couple in my household).

We will program this game (sans Guitar and Hero) and you can see it in the video below. The player must press a correct key (*left*, *down*, or *right*) whenever the target crosses the line. Pressing it to early or too late counts as a mistake. Of course, the faster the targets go, the harder it is to respond on time and with a correct key. As I wrote above, we will use the 3-up-1-down staircase procedure to control for that.

As per usual, we will take gradual approach:

- Boilerplate code
- Create a class for individual moving targets
- Create a timed-response task class that will create them (using cool generators), dispose of them, check the response, and adjust staircase.
- Add bells-and-whistles like score and time limited runs.

---

[2]It is the usual paradox that in order to optimally measure a threshold condition for a particular task, you should measure at or around the threshold. But if you already know where to measure, you don't need to measure.

13.3   Boilerplate

Create our usual boilerplate code in *code01.py*:

- Create file with basic settings (e.g., window size, I've picked 640×480 but choose whatever looks good on your screen) to which you can add later on.
- Import what is needed from PsychoPy.
- Create a window.
- Create our usual main game loop with `gamover` variable, flipping the window, and checking for *escape* button press.

Put your boilerplate code into *code01.py*.

13.4   Target and TimedResponseTask classes

Our main work horse will be `TimedResponseTask` class. It will spawn a new random `Target` at random intervals (which will depend on speed), pass speed information to moving targets, and remove targets, once they disappear below the screen. The `Target` class will use visual.Rect [3] with some extra bells and whistles to make it appear at the right location, move at the right speed, change its line color (indicating a correct response), compute whether it is already off the screen, etc. We will start with a single target first.

13.5   Target class: static

First, create a `Target` class: a colored rectangle in one of the three positions that starts at the top of the window and moves down at a specific speed. Its constructor should take PsychoPy window as a parameter (you will need it to create the rectangle as an attribute), position index (`ipos`, from 0 to 2), speed (`speed`, in `"norm"` units *per second*), and common settings (`settings`, a dictionary with target-specific settings from our settings file) . The only thing we need to do right now in the constructor is to create a color rectangle (see below) and store both `ipos` and `speed` as attributes for later use. In addition, define a `score` attribute and set it to `None`. This will hold the score the participant got for this target and `None` means that it has not been responded upon yet.

The second parameter — position index — determines the horizontal position of the target and its color (to make targets more fun and distinct). For my code, I have decided to make rectangle 0.4 norm units wide and 0.1 norm units high. The leftmost *red* rectangle (for `ipos` 0), is centered at -0.5, the middle *green* one is dead center, and the rightmost *blue* rectangle is centered at 0.5. I've defined all these in my `settings.json` file under `Target` group. Think about how you can compute both color and position for a target from `ipos` and

---

[3]Inheriting from `visual.Rect` would've make things simpler but, at least in PsychoPy 2021.1.4, it does not work due to so technical glitch.

`settings` without using if-else statements. Also, think about the y-position of the rectangle, so it appears right at the top of the window.

The second method you need is `draw()` which simply draws the rectangle[4]. Test it by creating a target at one of the position (or three targets at all three positions) and drawing them in the main loop. You should get nice looking but static rectangle(s).

Put updated code in code02.py_ and create the class `Target` in a separate file.

13.6   Target class: moving

Our targets fall down at speed defined by their `speed` attribute. Later on, we will change that attribute dynamically to speed up or slow down their fall.

For the actual falling down, implement a new method, call it `fall()`, that will update target's position on every frame. The speed is in `norm units per second`, thus, to compute the change in vertical position you also need to know the how much time *in seconds* has elapsed since last position update. The simplest way to do this is by using a Clock class. You create it as an attribute in the constructor and then, in the `fall()` method you use its current time to compute and apply a change in vertical position of the rectangle. Don't forget to reset the clock after that!

Include `fall()` method call in the main loop and see how the target falls. Experiment with falling speed!

Put updated code in *code03.py* and update the class `Target`.

13.7   Iterator/Generator functions

In the next section, we will create a `TimedResponseTask` class that will generate targets at a random location and after a random interval. We can, of course, do it directly in the class but where's fun in that?! Instead, we will use this as an opportunity to learn about iterator/generator functions. An iterator is a *function* that uses `yield` instead of `return` statement to, well, *yield* a value. It *yields* it, because the function itself *returns* an iterator object that you can iterate over in a for loop or via `next()` function. Importantly, `yield` "freezes" execution of the function and the next time you call the function *it continues from that point* rather than from the start of the function. Once you reach the end of the function, it automatically raises `StopIteration()` exception, so you don't need to worry about how to communicate that you ran out of items. It may sound confusing but it is really simple. Here an example to illustrate this:

```
def iterator_fun():
    yield 3
```

---

[4]Again, inheriting from `visual.Rect()` would have taken care of this for you.

```python
    yield 1
    yield "wow!"

# function returns an iterator, not a value!
print(iterator_fun())

# iterating via for loop
#> <generator object iterator_fun at 0x000001BA5D8CD740>
for elem in iterator_fun():
    print(elem)

# iterating via next(), note you use an iterator object
# that function returned, not the function itself!
#> 3
#> 1
#> wow!
an_iterator = iterator_fun()

# now you can use an_iterator to get a next item from it
print(next(an_iterator))
#> 3
print(next(an_iterator))
#> 1
print(next(an_iterator))
#> wow!

# next call will raise an exception StopIteration(), so I do not run it here.
print(next(iterator_var))
```

This format makes writing iterators very easy, just `yield` whatever you want in
an order you want and Python will take care of the rest. You can also `yield` in
a loop, inside an if-else statement, etc. Look at the code below and figure out
what will be printed out before running it.

```python
def iterator_fun():
  for e in range(4):
    if e % 2 == 1:
      yield e

for item in iterator_fun():
  print(item)
```

For our `TimedResponseTask` class, we will need two *generators*. They are *generators* rather than *iterators* because both will be endless. One that generates
a random delay until the next target and one that generates a random target
position (0, 1, or 2). Implement both in a separate file (I called it *generators.py*).

The `time_to_next_target_generator()` function should take a tuple of two float values, which define shortest and longest allowed delays, as a parameter and *yield* a random number within this range in an *endless* loop. We need the endless loop (`while True:` will do) because we do not know how many values we will need, so we just generate as many as needed.

The `next_target_generator()` will be a bit more interesting. It can just return a random.choice from 0, 1, and 2 but where is fun in that? Instead, we will make it a bit more complicated to ensure that all three targets appear equal number of times within *3N* trials, where *N* will be a parameter of the generator function. This would ensure random, reasonably unpredictable but balanced targets in the short run. Remember, in the long run random choice will always give us a balanced uniform distribution but there is not such guarantee for the shorter runs of a few trials. First, you should create a list where each target appears N times (think how you can do it using range(), list() and *). Then, create an endless loop (again, we don't know how many values we will need) in which you 1) shuffle elements of the list, 2) yield one element at a time via for loop. Once you run out of elements, you shuffle them again and yield one by one again. Then repeat. And again, and again. Endless loop!

I would suggest creating and testing both function in a Jupyter notebook first and then putting them in a separate file (e.g., *generators.py*). Be careful if you decide to use a for loop instead of `next()` for testing. Remember, both a generators and will never run out of items to yield for a for loop!

Put both generators into *generators.py*.

13.8   TimedResponseTask class

Now we are ready to create the `TimedResponseTask` class. For our first take, it will create targets at a random location (`next_target_generator()`) after a random interval (`time_to_next_target_generator()`) plus take care of moving and drawing all of them. More bells and whistles (disposing of targets that went past the screen, changing the speed, checking response validity, etc.) will come later.

For the constructor, we definitely need PsychoPy window as a parameter, because we need it every time we create a new target. In addition, we need to pass a dictionary with settings for the task (initial speed, a tuple with range for time intervals between targets for `time_to_next_target_generator()`, and number of target repetitions for the `next_target_generator()`) and a dictionary with settings for the `Target` class (we need it every time we create a new target). We will use these parameters beyond the constructor, so save them as attributes. Plus, create an attribute `targets` and initialize it to an empty list (we will store `Target` objects in it), and create attributes for both generator objects using the appropriate parameters. Also create a `speed_factor` attribute and set it to 1. We will use it later to control both the speed of motion and

how frequently the targets are generated. The higher is the factor, the faster targets move and the shorter is the interval to the target and vice versa. Finally, we need a Clock that will count the time to the moment when we need to generate a new target (`new_target_timer`) and an attribute that will hold that time (`time_till_next_target`). Initialize the latter to the `next()` item from time-to-next-target generator (remember, you need to use the attribute, which is a generator object that function returned, not the function itself).

Now we need to add three methods `draw`, `update`, and `add_next_target`. The first one is easy, it simply draws all `targets` in a for loop. The second is also easy, it makes all targets `fall` plus, after the loop, it should call `add_next_target` method. The `add_next_target` method should check whether the elapsed time for `new_target_timer` **times the** `speed_factor` (as the speed increases, the time to the next target goes faster) has exceeded the `time_till_next_target.`If that is indeed the case, create a new random target (get the `next()` position from the position generator and remember to pass `speed` *times* `speed_factor`!), add it to the list of targets, reset the timer and get new `time_till_next_target` using `next()` item from the time generator.

In the main file, create `TimedResponsTask` object (use a name you like) and call its `draw` and `update` methods in the main loop. You should see targets appearing at random and falling down consistently.

Put updated code in *code04.py* and create the class `TimedResponseTask`.

13.9   Disposing of targets

Currently, our targets keep falling down even when they are below the screen. This will not affect the performance *immediately* but it will be taxing both memory and CPU, so we should dispose of them. In the `Target` class, create a new read-only (computed) `@property` called `is_below_the_screen` that returns `True` if the upper edge of the target is below the lower edge of the screen (`False` otherwise, of course and you definitely do not need if-else!).

Next, in the `update` method of `TimedResponseTask`, add a second loop (or modify the existing loop) where you delete any object that `is_below_the_screen`.

For debugging, run the main code, wait until at least one target falls below the screen, put a break point and check `targets` attribute. Its length should match the number of visible targets, not of the total generated targets.

Update classes `Target` and `TimedResponseTask`.

13.10   Finishing line

Add a new visual attribute to the `TimedResponseTask` that is a horizontal line. The task of the player will be to press a corresponding key whenever a target crosses (overlaps with) the line. For now, create it as an attribute in the constructor (pick the vertical location you like) and draw it inside the `draw()` method.

Update class `TimedResponseTask`.

13.11   Response

Now the real fun begins! We will allow a player to press keys and check whether a corresponding target is on the line. For this, we need new methods for both `Target` and `TimedResponseTask` classes. For the `Target`, implement a new method class `overlaps()` that will take a vertical position (of the finishing line) as the only float number parameter. In the method, first you check that the `score` attribute is `None`. If it *not* `None` that means that the player already responded on to the target and they are not allowed to respond to it twice. If it is `None`, compute a score using the following formula:

$$score = int\left(10 - 10 \cdot \frac{|y_{target} - y_{line}|}{h_{target}/2}\right)$$

where y_{target} is the vertical center of the target, y_{line} is the vertical position of the line (you get it as a function parameter), h_{target} is height of the target, || means absolute value (use fabs function from *math* library for that), and `10` is an arbitrary scaling factor (you can use any integer). Study the formula and you will see that score is 10 if the target's center is right on the line but decreases linearly with any displacement. Once the target is off the line, the score becomes negative. We convert it to `int`, because we want simple scores (floats look messy for this). Compute the score and store in a *temporary local variable*. If the value is *positive*, that means success, so you should store this value permanently in the `score` attribute, change line color of the rectangle to white (to show the player that they got it right), and return `True` (yes, target does overlap with the line!). For all other outcomes, you return `False`. This means that either the response was already made or the target does not overlap with the line.

In the `TimerResponseTask` class, we need a new method `check()` that will take position of the target based on the key press (so if a player pressed *left* key, the position will 0, *down* is 1, and *right* is 2). Loop over targets and if a target's position (`ipos` attribute) matches the position of the key press (parameter of the function) *and* target overlaps with the line (the `overlaps()` method returns `True`), return the `score` attribute of that target. Note that the condition order is important here! You need to check for the overlap *only* if target position matches the key. If you ran out of targets to check that means that the player

pressed a wrong key or at the wrong time, so you should return `0` (means "mistake").

In the main loop, add `"left"`, `"down"`, and `"right"` to the key list of getKeys() call. Then, if any of these three keys are pressed, translate that into a position, respectively, 0, 1, and 2 (think how you can do it without if-else via a dictionary), and call the new `check` method of the `TimedResponseClass`. Test the code, targets' edges should turn white, if you time your key press correctly!

Put updated code in *code05.py*, update `Target` and `TimedResponseTask` classes.

13.12 Score

Playing is more fun when you can see how well you are doing. Let us add a simple score indicator that is updated with response score. You already know how you can do it via TextStim stimulus but you also already know how you can inherit from a base class and extend its functionality. This is what we will do here, as the class will record and draw the score (that part is covered by the inheritance).

Create a new class (I have called it `ScoreText`) that inherits from TextStim. In the constructor, you need to create an integer attribute that will hold current `score` and initialize it 0. Plus, call ancestor's constructor via `super().__init__(...)` to initialize and place the text stimulus (I've picked top left corner). Think about parameters that the constructor and ancestor's constructor need.

Next, we need to update the score (both its numeric form *and* the text that we draw) every time participant presses a key. We could implement the code *outside* of the class but that is a fairly bad idea, as it puts class-related code elsewhere. We could also implement a "normal" method, e.g., `add()` that will take care of that. Instead, we will implement a *special* method **iadd** that allows to "add to" the object. It takes a single parameter (in addition to the compulsory `self`, of course), performs "adding to the self" operation (whatever that means with respect to your object, can be mathematical addition for an attribute, concatenation of the string, adding to the list, etc.), and **returns back the reference to itself**, i.e., returns `self` not a value of any attribute! Here's how it works:

```python
class AddIt():
    def __init__(self):
        self.number = 0

    def __iadd__(self, addendum):
        self.number += addendum
        return self # important!!!
```

```python
adder = AddIt()
print(adder.number)
#> 0
adder += 10
print(adder.number)
#> 10
```

Implement that special method for your class, so we can do `score_stim += timed_task.check(...)`. Remember, you have to update both numeric *and* visual representations of the score in that method! Add the score to the main code.

Put updated code in *code06.py*, create `ScoreText` class.

13.13 Staircase

We will implement the staircase as part of the `TimerResponseTask` class, so it can speed up and slow down itself. For this, we will need an attribute that counts number of *consecutive* correct responses (I, typically, call it `correct_in_a_row` or something like that). Create and initialize it to zero in the constructor.

Next, create a new method `staircase()` that will take a single parameter (beyond `self`) on whether the response was `correct` or not. If it was, increment `correct_in_a_row` by one and check whether it reached 3. If it did, increase the `speed_factor` by *multiplying* it by some chosen factor (I've picked 1.3) and resetting `correct_in_a_row` to 0. This is equivalent to using a logarithmic step as our `speed_factor` is adjusted as a fraction of its magnitude. Alternatively, if the response was not correct, *divide* `speed_factor` by the same number (e.g., 1.3, so slowing things down) and again, reset `correct_in_a_row` to 0. After that, loop over all targets and update their speed based on `speed` and `speed_factor` attributes.

You need to call this method inside the `check` method, think then and how.

Update `TimedTaskResponse` class.

13.14 Limiting time

Let us add a competitive edge by limiting the run time to 20 seconds (you can pick your own duration, of course, and you definitely want to be a setting). Create an additional outer loop, so that the game can be played many times over. Once the round is over, show the latest state (redrawing all game objects) plus the "Round over" sign and wait for the player to press either *escape* (then you exit the game) or *space* (to start the next round). Remember to recreate all game objects anew for the next round (or create a `reset` method for all of them).

Put updated code in *code07.py*, create `ScoreText` class.

13.15   Using PsychoPy's StairHandler

Now that you know how to program a very basic staircase, let us use its much more flexible implementation by PsychoPy via StairHandler class. We will use so as to replicate the staircase that we already implemented. However, it is capable of much more and PsychoPy has implementation for other adaptive methods, such as parametric Psi or Quest approaches. I strongly recommend consulting the literature to decide which method is best suited for your experiment and then relying on PsychoPy's implementation in your code.

We will need to modify our `TimedResponseTask`, so let us create its twin `TimedResponseTask2` (or `TimedResponseTaskPsychoPy`, if you find that more intuitive). Simply copy-paste the entire code, modify the name, import and use it in your *code08.py* code. Make sure everything works just as before (because you did not do anything beyond making a carbon copy).

Now let us make use of the StairHandler in `TimedResponseTask2`. Drop `correct_on_a_row` attribute and create a StairHandler as `stairhandler` attribute instead. You need to specify `startVal` which is the initial value for the `speed_factor`, thus use whatever value you had previously. StairHandler uses `nUp=1` and `nDown=3` by default. This matches our custom staircase, so theoretically you can use defaults by omitting these parameters. However, for the sake of code's readability, do specify these explicitly. Our steps were logarithmic, so use `stepType="log"` and a single fixed `stepSizes=-0.1`. The magnitude of `-0.1` correspond roughly to the step that we used in the custom staircase and we need the negative sign because StairHandler *increases* the staircase level following an incorrect response. In our case, we want an exact opposite, *decreasing* `speed_factor` tp slow targets down. Hence, the negative sign that turn increase into a decrease. Finally, StairHandler will terminate after it reaches either desired number of trials (`nTrial`) or reversals (`nReversals`, changes from correct to incorrect responses or vice versa). These are the settings that would typically determine length of a single block/run in the real experiment. However, we limited our rounds by *time*, so we only need to make sure that the StairHandler does not run out of trials before the run is over. Thus, specify some very large number (e.g., 1000) for both these parameters.

Once you created `stairhandler` attribute, it is ready for use via `next(staircase)`. Call it the first time in the constructor and assign the value it returns to `speed_factor` attribute (should be whatever `startVal` you assigned to it but do put a breakpoint and double-check!)

Next, we need to modify our `staicase()` method making it much simpler. First, remove the `if correct: ... else: ...` code but leave targets' speed adjustment code intact (we still need it!). Then, let `stairhandler` adjust itself

via addResponse() method using an information on whether the response was correct (you already have a parameter with exactly that information). Finally, get the next `speed_factor` exactly the same way as in the constructor. Done!

Put updated code in *code08.py* using `TimedResponseTask2`.

Your program should run very much like before but now you have many more opportunities to make it more flexible at little cost for yourself (look at StairHandler settings) and to log it via one of `saveAs` methods.Let us do the latter, save staircase logs via saveAsText() after a run is over. Figure out a way to generate a unique filename for each run, so that the logs will not be overwritten.

Save staircase logs in *code09.py*.

13.16 This is just a start!

As per usual, think about how you can extend the game. A clock showing the remaining time is definitely missing. Auditory feedback would be nice. More positions? Random colors to confuse a player?

Chapter 14

The Snake

This chapter marks the beginning of the second semester in the course, therefore we will use our first game to refresh your knowledge of Python acquired previously. We will use classes (and their instances, objects), lists, conditional statements, loops, and external setting files to program a classic game of Snake. Consults previous chapters for any information you need to complete the game.

14.1   Snake game: an overview

Today, we will program a good old classic: the snake game! The story is simple: you control a snake trying to eat as many apples as you can. Every time you consume an apple, snake's length increases. However, if you hit the wall or bite yourself, the game is over (or you lose one of your lives and game is over once you run out of lives).

Here is how the final product will look like.

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

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

---

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

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

14.2   Minimal boilerplate

We need the usual boilerplate code to get us going:

- Create a PsychoPy window (hardcode its size, as we will compute it from settings later on).
- Wait for any key press.
- Close the window.

Put your code into *code01.py*.

Do not forget to put the multiline comment at the top of the file, describing what we will be doing.

14.3   Fancy boilerplate

In the game, our snake will be composed of square segments and move on a grid made out of squares. Thus, it would be much easier to define our window size based on the size of the grid (in squares) and that of individual squares (in pixels). This way we can figure out window's height as

$$window\ height\ [in\ pixels] = window\ height\ [in\ squares] \times square\ size\ [in\ pixels]$$

Note that the latter parameter determines how the game looks, double the size of the square in pixels and that will double both width and height of the window.

Now, let us think about the *units* that we will use in our game. Recall that PsychoPy has five different units for size and position. So, which units should we pick to make it easier to draw squares? Reread section on units and think which units you would pick before continuing.

———————————————

What is your decision? My suggestion would be `"norm"` units as they make sizing squares easy, as long as we use a suitable aspect ratio for the window itself. Good news is that the formula above automatically ensures the correct aspect ratio (think why this is the case).

Before we start coding the window size computation, create a settings file using the format of your choice that contains `grid size [in squares]` (with a list of two values or you could define it as `grid width [in squares]` plus `grid`

height [in squares]) and square size [in pixels]^2. I would suggest to start with a 30×20 grid as it should give us enough space to try things out but you can always increase the resolution of the game later. As for the square size, 20 pixels should be enough (but decide based on your screen resolution).

Add the code for loading the file into settings variable and then use the settings to compute the window's size. Test that it works correctly by doubling (or halving) the size of the grid or of the square size.

Put your code into *code02.py*.

14.4 Fancier boilerplate

The code for computing window size belongs to the business logic of setting up a window and, therefore, should be part of a window class code. Therefore, let us define a new GridWindow class that will inherit from PsychoPy Window class but modify the constructor to perform a window size computation. The new __init__() method should take grid size and square size as parameters, compute an actual size of the window in pixels, and call the __init__() method of the *parent* Window class to finish the setup (if you forgot how to call parent's methods, take a look at how we implemented Flappy Bird class by inheriting from ImageStim). You already have all the code you need but in the main script, so the only thing you are doing is tidying things up by moving it into the constructor where it belongs and adjusting it for use of parameter and parent methods.

Put your GridWindow class into a separate file (gridwindow.py?).

Modify the code and save main script into *code03.py*.

14.5 Adding a square

As I already wrote, our game will be made of squares. Snake is made of squares. Grid it lives on is made of squares. An apple is a square^3. Which means we need to know the size of that square in the units of the window and we need to know where within the window each square should go based on its coordinates on the grid.

Computing the size of the square in units of window size is easy, so let us start with. If you used "norm" units for the PsychoPy window, we know that both its width and its height go from -1 (bottom) to 1 (top). We also know that we have to fit in settings["grid width [in squares]"] squares horizontally andsettings["grid height [in squares]"] squares vertically (If you

---

^{2}I always add units in square brackets as a reminder to my(future)self. One can deduce units from code itself but documenting them explicitly makes your life much easier.

^{3}We will use a bitmap to make it look like an actual apple but it will need to be a square bitmap, so at the end the apple will still be square

encoded grid size as a list of `[width, height]` then `settings["grid size [in squares]"][0]` and `settings["grid size [in squares]"][1]`). Compute the square width and height in the window units in the constructor method of the `GridWindow` and store it for later use as a tuple in `square_size` attribute. Also, store the grid size in an attribute (e.g., `grid_size`) as it will be useful for snake and apple classes later. Once you implemented the code, use debugging to check that numbers make sense. E.g., if your grid is 20 squares tall then each square should be 0.1 `"norm"` units tall.

Next, let us create a method that maps a position on the grid to a position in the window. This way, we can think about position of the snake or apples in terms of the grid but draw them in window coordinates. Create a new method `grid_to_win()` in the `GridWindow` class that takes a single parameter with a tuple of integers (`x_index, y_index`) with grid position (I called this argument `ipos`, here and in all other places, I am not counting the compulsory `self` parameter) and returns a tuple of floats (`x_pos, y_pos`) with the coordinates of the *center* of the square in *window* coordinates. Use list comprehension to simplify the code.

Take a look at the drawing below to see the geometry of the computation. The red text shows location of red points in *window norm* units, whereas the black font shows location of a square in *grid index* units. Note that you need to compute where the *center*(!) of the square should go.
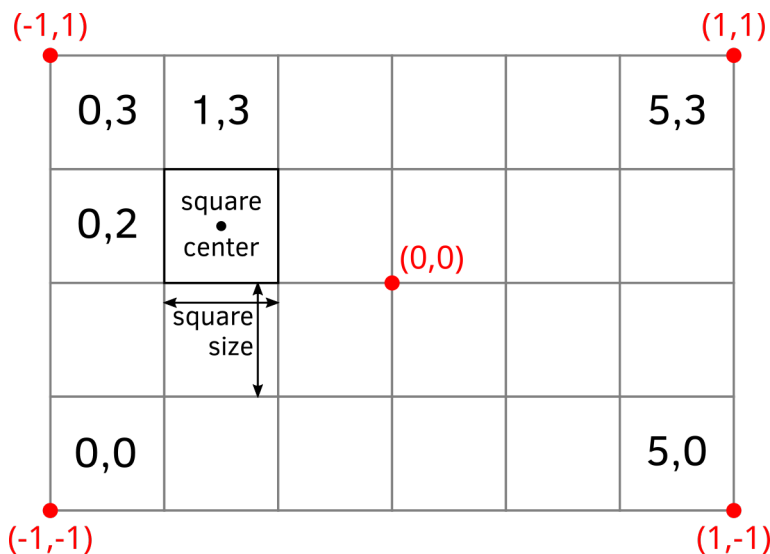

Figure 14.1: Grid versus window coordinates

Remember to document the method following NumPy docstring format.

Now, test this method by creating a square (you now get its size from the

`GridWindow` attribute(s) that you created) in the main script and placing it at different locations on the grid. Run the code several times, using different grid indexes or adding several squares to check that it works as intended. I.e., for a 30×20 grid, a grid coordinate `0×0` should place a square at the bottom-left corner, whereas `14×9` should put it almost at the center.

Put your code into *code04.py*.

14.6 A snake segment

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

Consider an individual segment. We need to 1) keep track of its location in *grid* coordinate system and 2) have its visual representation — a square, just like the one you create during the previous exercise — positioned in *window* coordinate system. Good news is, you already have a method of the `GridWindow` class that maps the former on the latter, so as long as you know where the segment is on the grid, placing its square within the window is easy. Thus, each segment has two pieces of information associated with it and we could represent it as a dictionary with two keys (`"pos"` (tuple of x and y in grid coordinates) and `"visuals"` (the square)) or, better still, as a class of it own.

Let us create a `SnakeSegment` class. Its constructor `__init__()` methods should take three parameters: 1) a PsychoPy [Window] object, 2) segment coordinates on the grid, 3) segment color. Save the grid coordinates into the `ipos` attribute (we need to keep track of it when the snake will start moving around) and reuse the code from the previous exercise when created a square. Store that square in `visuals` attribute and add a `draw()` method that simply calls the `draw()` method of the square[5].

Test the class by creating and drawing a snake segment in the main script. Define snake segment color in settings, create it inside a separate section `"snake"` as it will add more parameters later on.

Put your `SnakeSegment` class into a separate file (*snaking.py*).

Modify the code and save main script into *code05.py*.

14.7 The snake

As already noted, a snake is just a list with snake segments. Create a new class `Snake`. Its constructor should take just two parameters: a PsychoPy

---

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

[5]It would be much simpler to inherit from Rect class directly but it does not work (raises an error), at least for the version 2021.1.4 that I use.

window object and a dictionary with settings (a `"snake"` section of our `settings`
variable in the main script). In the constructor, create a single segment and put
it into a *list* attribute `segments`. We will have just one attribute for now but
using a list already helps us in the future. Place that only segment roughly at
the center of the grid. Use `grid_size` attribute of the `GridWindow` to compute
it (you will need to use a floor division `\\` for that). You should also store the
window in an attribute for a later use, as we need it every time we create a new
segment.

In addition, implement a `draw()` method that draws all `segments` in a loop.
Test your code by creating and drawing a snake in the main script.

Put your `Snake` class into a separate file (*snaking.py*).

Modify the code and save main script into *code06.py*.

14.8   Get a move on!

Now we need to understand how we will move the snake given that it consist
of many segments. Assume that we have a five segment snake that moves up,
as in the picture below (segments are color coded to make it easier to see who
goes where).

Time 1   Time 2
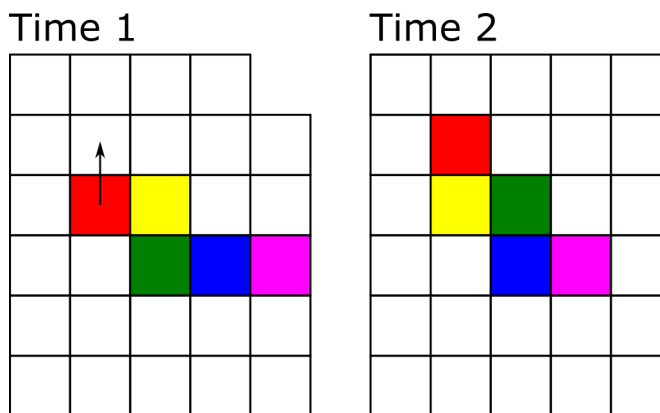
Technically, we need to move each segment to a new position. The very first
"head" segment moves to the position above the snake. The second segment
moves to where the head was before. The third moves into the previous position
of the forth one, etc. We *can* implement movement like that but instead we will
utilize the fact that, unless colored as in the figure above, all segments look
identical. Look at the uniformly colored snake below. We can "move" it by
adding a new segment at a new location (growing a new head at the bitey end
of the snake, marked as red) and clipping off the last tail segment (marked by
the cross), so that the previously penultimate segment becomes the tail. The
rest can stay where they are, saving us a lot of hustle when the snake is long!

Time 1

Time 2
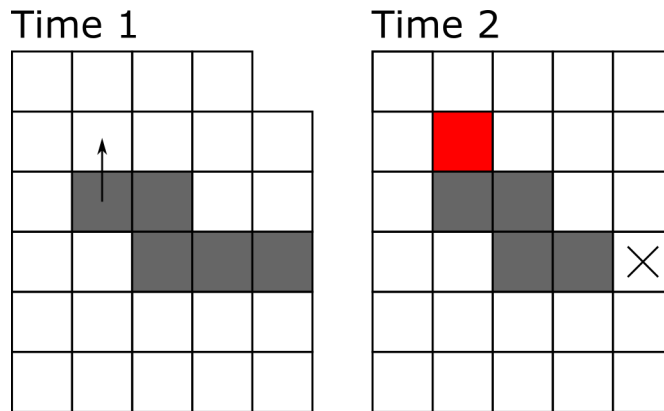
In the program, we list.insert(index, value)[6] the new head segment at index 0 and we remove the tail via list.pop().

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

The `grow()` method should take just a single tuple parameter `dxy` with a direction of snake's movement. I.e., it will be `(-1, 0)` for leftward moving snake, `(1, 0)` for right moving, `(0, -1)` for up, and `(0, 1)` for down. Inside the method, you compute the coordinates for the new head segment based on the position of the current head (the very first element of the snake) and the direction of motion (`dxy`), create a new snake segment at that location, and insert it before all other elements.

The `trim()` method is even simpler as its just pops the last segment (trims the tail).

To test these functions, create a snake and call its `grow()` method a few times using the same or different directions. Check visually that it works, i.e., that there are correct number of segment (initial head + as many segments as `grow()` method calls) and that the snake was growing in a correct direction. Combine `grow()` and `trim()` calls to visually check the snake as well.

Update `Snake` class adding `grow()` and `trim()` methods.

Test it in *code07.py*.

---

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

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

14.9   The game loop

Add a main game loop using a `user_abort` variable, so that it runs for as long as the `user_abort` is `False`. Inside the loop, draw the snake and check for keys without pausing the game. If a player pressed `"escape"`, change `user_abort` to `True` (the player decided to quit the game). If a player pressed `"space"` key, grow the snake in the direction of your liking and trim it. Thus, every time you press a *space* key, the snake should move one square in the direction of your choice. Test the code!

Add main loop in *code08.py*.

14.10   Self-motion

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

Instead, we should decide on snake's speed, e.g., 4 squares per second[8], and define it as a new parameter in settings. In the `Snake` class constructor, use this parameter to compute a new attribute `step_duration` that expresses the time needed to elapse before moving to a new square (i.e., interval between calls of grow/trim methods in seconds)[9]. Why two variables that express the same information (interval is just an inverse of speed)? We could define `step_duration` directly in the settings but for a human it makes it harder to understand just how fast the movement will be. At least for me, a speed of `4 squares per second` is easier to comprehend than `0.25 second per square`. Thus, in cases like these, I prefer to have two variables, one human-oriented (in settings) and one computer-oriented (in the actual implementation, computed via a unit conversion). Remember, it is not just about writing a working code, it is about writing a code that is easy for a human to understand.

Also in the constructor, add a new `movement_clock` attribute creating either a clock or a countdown timer object[10] Then add a `reset_clock()` method

---

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

[9]Do it in the constructor, always define all attributes in the constructor, even if they are set to `None`.

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

and a `can_move` computed property. The former should reset the clock or the countdown timer (whatever you have picked). The latter should return `True`, if the timer ran out and the snake can move to the next square (in this case you need to reset the timer!), or `False` if that is not the case. In the main script, call `reset_clock()` method before the main loop and inside of the main loop call `grow()` and `trim()` methods *only* if the snake `can_move`.

Test your code by setting different snake speed.

Update `Snake` class adding timer attributes and methods.

Test it in *code09.py*.

14.11 Describing direction using words

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

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

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

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

However, this approach introduces a lot of repetitive code and does not scale particularly well. Instead, we can use a dictionary (let us call it `DIRECTION_TO_DXY` as it is another constant) with `"up"`, `"right"`, etc. string as keys and tuples `(0, -1)`, `(1, 0)`, etc. as values. This way, we can use current value of the `direction` variable as a key to get the `(dx, dy)` pair from `DIRECTION_TO_DXY` dictionary. Do this translation directly in the `grow()` method call rather than creating a temporary variable as in the if-elif examples above. Test the translation by changing `direction` and checking that the snake moves accordingly.

Use `direction` and `DIRECTION_TO_DXY` in *code10.py*.

14.12   It is all about control

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

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

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

Add `NEW_DIRECTION` and implement steering in *code11.py*.

14.13   Turning the hard(er) way

Let us implement the same "figure out new direction" code in a more complicated way. The purpose of the exercise is to challenge you, show you new methods of the list, and to demonstrate how you can think about a change of the direction as moving through the list. Implement it as a function called `compute_new_direction()`, which will take two parameters (current direction and pressed key) and will return the new direction of rotation. You can put it above the main script or, better still, put it into a separate *utilities.py* file. However, in cases like these, I find it useful to play with the code in a Jupyter notebook before turning it into a function, so you could try this approach instead of debugging.

Here is the idea. Imagine that you have a list `["left", "up", "right", "down"]`. For this list, rotation clockwise would correspond to moving through the list to the right (assuming that you jump to the beginning once you move past the last item). Conversely, rotation counterclockwise corresponds to moving to the left (again, assuming that you jump to the end of the list, once you went past the first item). As you see, rotation is expressed as a very intuitive "motion through the list".

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

Create 'compute_new_direction() in *utilities.py*.

Use it instead of a dictionary look-up in *code12.py*.

14.14   Hitting the wall

We can control the snake but, at the moment, you can steer it off the screen or make it go through itself. Let us fix the former!

Add a new computed property `hit_the_wall` to the `Snake` class. The method should check whether the head of the snake (which segment is it?) is still within the grid that we defined (how do you check for that?). It should return `True`, if the head of the snake is outside of the grid limits (so, it is true that the snake hit the wall) or `False`, if it is still inside.

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

Test that the game quits after you successfully steered the snake into the wall.

Add `hit_the_wall` property to the `Snake` class.

Use it in *code13.py*.

14.15   Is this the snake?

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

Add a new method `is_inside()` that takes a tuple with a grid location as a parameter and returns a logical value whether that grid location is occupied by the snake (i.e., by one of its segments). Document the function! Test the method via debugging. Calling it immediately after creating the snake with a location of the head (the only segment) or with a different location. Store a return value into a temporary variable and put a breakpoint at an appropriate place to check its value (or use a debug console).

Add `is_inside()` method to the `Snake` class.

Test it in *code14.py*.

14.16   An inedible apple

Let us add that highly desirable fruit: the apple! We will use an apple bitmap[11], so we will create a new class as a descendant of ImageStim class. Its constructor should take `GridWindow` and `Snake` as parameters. In the constructor, you need to find a random grid location that is not occupied by snake, store it in an `ipos` attribute and call the parent's constructor to initialize the image with an appropriate size and location (again, check Flappy Bird class or `GridWindow` class, if you forgot how to do this).

The only tricky part is finding the unoccupied location. There are many different strategies for doing this. The simplest one is to generate a potential location at random, check whether it is occupied by a snake (`is_inside()`?) and keep doing it until you find an unoccupied one.

---

[11]Created by Jess Wiechler.

In the main script, create an apple and draw it in the main loop. For a moment, the snake cannot eat it and will pass through it but we will fix this shortly.

Create `Apple` class in *apples.py*.

Use the apple in *code15.py*.

14.17   Eating an apple

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

You need to add a conditional statement that if the snake's head *is* on the apple, you should not trim the tail but create a new apple. What should you do, if there is no apple at that location?

Make snake eat apples in *code16.py*.

14.18   Eating yourself

Once our snake grows beyond four segments, it has an opportunity to bite itself[12]. For this, we need to check that, *after* the snake moved, its head is *not* at the same location as one of the segments of its body. Create a new property `bit_itself` that returns `True` or `False` based on whether that is the case. The property is very similar but not identical to `is_inside()` method you implemented earlier. What is the critical difference and why cannot you simply reuse that function?

Once you implemented `bit_itself`, you should check for that it does lead to the end of the game.

Implement `bit_itself` property.

Use it in *code17.py*.

14.19   Bells and whistles: score

Now that we have a fully functional game, we can start adding non-essential but nice-to-have features to it. The first one will be the score. Create a new class `Score` that inherits from TextStim and has an extra attribute `score` (initialized to 0 in the constructor) and a method `plus_one()` that should increase the score by one and update the `text` attribute. As with an apple, initialize the TextStim part by calling parent's `__init__()` method using appropriate parameters. I

---

[12]Why at least five? Draw it on the grid and figure out whether it can eat itself with just four segments.

decided to put it at the top of the window and it reads **Score: XXX** but you are free to do it as you feel fit.

In the main script, create a `Score` object, draw it when appropriate and increase the score every time the snake eats an apple.

Create `Score` class in *scoring.py*.

Use it in *code18.py*.

14.20   Bells and whistels: three lives

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

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

Put your updated code into *code19.py*.

14.21   Bells and whistels: press space to start the round

At the moment, our round starts immediately. It would be friendlier, if the player would start it themselves. Before each round, draw *all* visuals (snake, apple, score) plus a text "Press SPACE to start" and wait until either a *space* or *escape* key is pressed. In the former case, the trial should start. In the latter case, the player should exit the game.

Put your updated code into *code20.py*.

14.22   Bells and whistels: showing remaining lives

Let us not just repeat the game three times but show the player how many lives they still have. Download the heart.png[13] and use it show remaining lives at the

---

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

top-left corner of the screen: three hearts in round one, two hearts in round two, and just a single heart in round three. You will need to use (ImageStim)[https://www.psychopy.org/api/visual/imagestim.html#psychopy.visual.ImageStim] for this. Think about the size of images and their location. Hint: I created a list of heart images using list comprehension and drawing hearts definitely requires a for loop.

Put your updated code into *code21.py*.

14.23   Bells and whistles: difficulty

At the moment, the difficulty of the game, the speed with which the snake moves, is fixed and the player has no way of choosing it. Let us create dialog that appears *before* we create the window and start the game that will allow the player to choose between *easy*, *normal*, and *difficult*[14]. I leave it up to you to decide which snake speeds correspond to each difficulty. You can have more than three options, if you want. However, they definitely should be define in settings file.

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

Put your updated code into *code22.py*.

14.24   Bells and whistles: blinking "game over" message

Once the game is over (but not when a player pressed *escape* key), show a blinking "Game Over" message *superimposed* over the final static game screen. Thus, you need to draw all the game objects and messages (but without moving the snake) plus you show a text message that is on for 0.5 second and off for 0.5 seconds until the player presses *Space* button. Hint: it should be a separate loop after the main game loop over rounds, logical variables and clocks/timers have definitely something to do with it.

Put your updated code into *code23.py*.

---

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

14.25   Bells and whistles: sounds

Download game-over-arcade.wav[15] and 8-bit-game-over-sound.wav[16]. Use the former whenever the snake dies and use the latter when the player runs out of lives. Note, no sound should be played if the player pressed *escape*.

You will need to use `Sound` class from sound module of PsychoPy. Important: use `Sound` class not a library-specific implementation such as `PTBSound` or `SoundDevice` classes. The PsychoPy will figure out which backend is the best (available at all) for you.

Put your updated code into *code24.py*.

14.26   The sky is the limit

Good job! Lots of coding but now you have a cool retro game!

---

[15]Downloaded from freesound.org and created by myfox14
[16]Also downloaded from freesound.org and created by EVRetro

Chapter 15

Moon lander

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

Here is the general outline of how we will proceed:

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

But first you will learn about context management and exceptions.

15.1   Boilerplate

As per usual, we will start with our usual boilerplate code. Create settings file that, for now, defines only the size of the window. Create `code01.py` with the usual boilerplate for loading settings, opening the window (its size determined by the settings), a main game loop (you can add a text message to make it look less plain) with a check for an *"escape"* button to exit the loop, and closing the window at the end. I am being so specific because next you will learn how to hide this boilerplate in a context manager.

Put your code into *code01.py*.

15.2   Context manager

On the one hand, context management is a frequently used feature in Python, particularly for file operations (you used it when loading settings from a JSON or YAML file). On the other hand, its full power that relies on a custom class implementation is rarely used. However, it can be very useful whenever the context of your programs is the same or very similar, as in case of the PsychoPy games that we programmed or typical PsychoPy experiments. In both cases, there is a fairly fixed structure of the program:

1. Initialization
   - get experimental settings by reading them from an external file
   - create PsychoPy window, logger for experimental results, mouse (if required)
   - initialize special devices such as response box, eye tracker, etc.
2. Actual experiment
3. Saving and cleaning up
   - save data logs
   - if required, close connection to special devices such as response boxes, eye tracker, etc.
   - close PsychoPy window

If you look at your code, you will realized that steps 1 and 3 remain pretty much the same throughout all the games that we programmed. Thus, we will create a context manager class that you can always reuse and which will hide away the boilerplate code.

Here is a reminded of how a context manager is used when working with files. First, how it works *without* a context manager: 1) you open a file and assign the object to a variable, 2) you work with it, 3) you close it. The latter is important to ensure that information was fully written into it and that you do not lock for file.

```
file = open("somefile.txt", "r")
# ... do something with the file, such as reading the entire file into a single variab
data = file.read()
close(file)
```

A better way is to use a context manager via a `with ... as ...` statement (again, this should look familiar by now):

```
with open("somefile.txt", "r") as file:
    file.read()
```

Note that now the `file.read()` is inside of the `with` block and there is no `file.close()` call. The latter is evoked automatically, once you run all the

code inside the `with` block and exit it. Although for this example the difference is minimal — a different way to assign a value to a variable and explicit versus implicit file closing — the second variant takes care of cleaning up, ensures that you do not forget about it, and allows you to concentrate on the important bits.

Here's how it works behind the scenes. A context manager is a class that implements two special methods `__enter__` and `__exit__`[Spoiler alert! This approach is called duck typing and we will learn more about in the next game]. The former creates and returns a context, which is whatever attribute or value you require, wheres the latter performs cleaning up that is necessary before exiting the context. Here is how we would implement a limited file context manager by ourselves:

```python
class FileManager():
    def __init__(self, filename, mode):
        """
        Stores the settings for use in __enter__

        Parameters
        ----------
        filename : str
        mode : str
        """
        self.file = None
        self.filename = filename
        self.mode = mode

    def __enter__(self):
        """
        What we need to do to create context:
          * Open the file and returns the object.

        Returns
        ----------
        File object
        """
        self.file = open(self.filename, self.mode)
        return self.file

    def __exit__(self, exc_type, exc_value, traceback):
        """
        What we need to do before destroying the context:
          * Close the file before we exit the context.
        """
        close(self.file)
```

```python
# and now we use it!
with FileManager("somefile.txt", "r") as file:
    file.read()
```

Note that `__exit__` method has extra parameters `exc_type`, `exc_value`, and `traceback`. They will be relevant for exception handling later on but you can ignore them for now.

Now is your turn! Create a `GameContext` class (in a separate file, of course) that will load settings (filename should be passed to the constructor), create a PsychoPy Window object of a given size upon entering the context, and close when the code exists the context. For now, you will need one attribute to store settings (call it `settings`) and one attribute for PsychoPy Window itself (use `win` as an attribute name). There will be a small but important difference relative to `FileManager` class in the example above. Here, we have two objects (attributes) that we would like to use inside the context: `settings` and `win`. We could return both as a tuple but this approach does not scale well. Instead, the **enter** should return the reference to the context object itself (reminder, reference to the current object is always in the `self` parameter of a method). This way you can always access either attribute via `context.settings` or `context.win`.

```python
with GameContext("settings.yaml") as context:
    # your usual code inside but
    # PsychoPy window is context.win
    context.win.flip()
```

Create `GameContext` class and use it in *code02.py*.

As you can see, the repetitive part is now hidden in the context class making it easier to concentrate on the main code. But the context manager has another ace up its sleeve: it makes handling exceptions (a.k.a. errors) and safe exiting much simpler.

15.3   Exceptions

When you are running an actual experiment, one of the worries that you have is "what happens to the data I have already logged if the program crashes with an error"? Not collecting a full measurement is bad but not keeping at least partial log is even worse, as you can still use it for analysis or as a guidance for future adjustments. Python, as other languages, has special mechanisms to handle exceptions that arise during the code execution.

Whenever an error occurs at a run time, it raises an exception: it creates an object of a special class that contains information describing the problem. For example, a ZeroDivisionError is raised whenever you try to divide by zero, e.g., `1 / 0` (you can try this in a Jupyter notebook). A KeyError is raised, if you using a dictionary with a wrong key, the code below will raise it:

```python
a_dict = {"a_key" : 1}
a_dict["b_key"]
#> Error in py_call_impl(callable, dots$args, dots$keywords): KeyError: 'b_key'
```

Similarly, an IndexError is raised, if you try to use an invalid index for a list, a NameError, if you are trying to access variable that does not exist, AttributeError when an object does not have an attribute you are trying to use, etc.

In Python, you use `try: ... except:...finally:` operators to anticipate and handle exceptions:

```python
try:
    # some code that might generate a runtime error
except:
    # code that is executed if something bad happens
finally:
    # code that is executed both with and without exception

# code that is executed ONLY if there were no exceptions or if an exception was handled
```

In the simplest case, you need just the first two operators: `try` and `except`. Create a Jupyter notebook (that you will submit as part of the assignment) and write the code that generates a division-by-zero error but is handled via `try...except....` In the `except` simply print out a message, so that you know that it was executed. Create another cell, copy the code and now check that the exception handling code is *not* executed, if the error is not generated (i.e., divide by some non-zero number).

Put exception handling code is cell of a Jupyter notebook.

Using `except:` catches *all* exceptions. However, this is considered a bad style (too general) and a linter will complain. Instead, you can be more specific and handle exceptions based on their class.

```python
try:
    # some code that might generate a runtime error
except KeyError as key_error:
    # code that is executed only if KeyError exception was raised
    # with exception information stored in the key_error object
except ZeroDivisionError as zero_division_error:
    # code that is executed only if ZeroDivisionError exception was raised
    # with exception information stored in the zero_division_error object
except:
    # code that is executed if any OTHER exception is raised.
```

Implement handling for `KeyError` and `ZeroDivisionError`, they should print out different messages to check that it works. Test it by generating these runtime errors with your code.

Put specific exception handling code is cell of a Jupyter notebook.

So far, you generated exception by causing runtime errors code but you can raise these exceptions yourself via raise operator. For example, instead of dividing by zero, you can `raise ZeroDivisionError()`[1]. Use it with you previous code, instead of an actual division by zero. Try raising other exceptions and see how your code handles them. Also check what happens if you have the first two specific exception handlers but no general `except:` and raise an NameError?

Use `raise` to test exception handling in a Jupyter notebook.

So far I have talked about exceptions as a way to alert about runtime errors. However, they can be used in a more general way to control the execution flow. We will use that side of exception in the next section when dealing with context.

15.4   Exception within context

`try..except...` operators provide a general mechanism for exceptions handling but what happens if an exception is raised inside a context? You can, of course, put a `try...except...` in the code itself, something you should do, if you are planning to handling *specific* exceptions. However, if an exception occurs in the code inside the context, Python will first *exit* the context, i.e., call the `__exit__` method, before handling it explicitly. Moreover, it will kindly put the exception information into the parameters `exc_type` (a class of the exception) and `exc_value` (an object of that class). This way, you can perform a proper clean-up (save data, close window, etc.) and then either handle an exception or leave it alone, so that it propagates further and can be handled by other pieces of your code (or it will stop the execution, if you do not handle it explicitly).

Here, we will use this mechanism not only for safe clean-up but also to make aborting an experiment (or a game) easy. In previous games with many rounds, you had nested loops that made aborting a game via *escape* key press awkward. You had to check it in the inner loop and then differentiate between a normal end-of-round and a used abort in the outside loop. We can make our life much easier via a combination of a context manager and a custom exception.

First, create a custom `GameAbort` class, which is a descendant of the `Exception` class. You do not need any code in it, even a constructor does not need to be redefined, so use pass statement for its body (you do need to have at least one line of code in the class). Next, you `raise GameAbort()`, if the player pressed *escape* key (do not forget to import `GameAbort` class, so you can use it in the main script). Finally, in the **exit** method of the `GameContext` manager, you should check whether `exc_type is GameAbort` (`exc_type` will be `None`, if no exception occurred) and, **very important(!)**, `return True` in that case:

---

[1]Confusingly, if you do not pass any additional parameters, you can also create the object *without* round brackets: `raise ZeroDivisionError`. I find this mightily confusing but this is fairly common, so drop brackets if it feels more natural.

```python
def GameContext:
    ...
    def __exit__(self, exc_type, exc_value, traceback):
        ...
        if exc_type is GameAbort:
            return True
```

That last bit `return True` informs Python that you handled the exception and all is good (not need to propagate it further). However, note that your *return True* meaning that any other code that handles the exit from a context must be *before* that statement. Now, you can safely abort your experiment from any code location, inside nested loops, functions, etc. In all cases, the exception will be propagated until the `__exit__` method, doing away with awkward extra checks.

Create `GameAbort` exception class,update `GameContext` class to handle it,use this in an updated main script in *code03.py*.

15.5 Create MoonLander class

In *moonlander.py*, create a new `MoonLander` class. It should have an ImageStim attribute (I will assume it is called `image`) that will contain the visuals of the ship created using ufo.png image. However, instead of hardcoding the filename, create a new group `"Lander"` in the settings file and add a new setting `"ship image : ufo.png"` (assuming you use YAML). Pass Lander-specific settings to the constructor and save them in an attribute (we will have more of them later).

Note that we do not want to inherit from the ImageStim directly, as we will have more visuals elements later on. Also, implement `draw()` method that should draw all visual elements of the lander (we have one for now, of course).

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

Create `MoonLander` class and use it in the main game loop. Put updated code into *code04.py*.

15.6 Randomize lander's position

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

Add `reset()` method to `MoonLander` class and use it in the main game loop.
Put updated code into *code05.py*.

15.7   Flying (but only down)

For the lander to fly, we must adjust its position or, more specifically, the
position of its image (`self.image.pos`) based on its speed. But before that,
speed itself must be adjusted based on the forces from gravity and thrusters
that act upon the lander. Accordingly, we need

1. A new setting that defines acceleration due to the gravitational force.
   Define in the settings file, call it `gravity [norm/sec^2]` (so, it is an
   acceleration in distance units of `"norm"` rather than in meters of the real
   world) and set to `0.0001`[2]
2. A new attribute `speed` that will contain horizontal and vertical velocity
   in norm units per second. Initialize to `[0, 0]` in the `reset()` but also
   assign some value (e.g., also `[0, 0]`) as linters do not like to see attributes
   that were never mentioned in the constructor.
3. A new attribute with a PsychoPy clock that will measure the time elapsed
   since the last position adjustment (this way we can compute our speed and
   acceleration "per second"). Do not forget to reset it in `reset()`.

Now, implement a method that will update lander position (call it, unimagi-
natively, `update()`). Inside, figure out the time elapsed since the last call (or
since the last `reset()`) and do not forget to restart the clock. Once you know
how much time has elapsed, you can adjust, first, speed based on acceleration
(only vertical speed based on gravity for now, we will worry about the horizontal
component once we implement thrusters) and, then, position based on speed.
Call it in the main loop and watch your lander fall out of the sky. Once it is
off the screen, press space and see it go again. Play with the `gravity` setting
to adjust the speed of falling to your liking.

Update `MoonLander` class for the effect of gravity. Use it in the main loop of
code06.py.

15.8   Vertical thurster

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

---

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

moonlander.py add `import pyglet` and then include the following code inside the constructor of the class.

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

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

Define a `vertical accelartion [norm/sec^2]` to be twice the gravity (but you can use some other number, of course) and update the `update()`[3], so that the total vertical acceleration is *verticalacceleration + gravity* if the the user is pressing *down* key (use `self.keyboard` and `self.key` to figure that out) but *gravity* alone, if not.

Test that the vertical thruster works (do you need to update the main code?)!

Update `MoonLander` class with a vertical thruster.

15.9 Horizontal thursters

Now implement the same logic, computing acceleration, speed, and position but for horizontal thrusters (define `horizontal acceleration [norm/sec^2]` setting and decide on its value yourself). Remember, the *right* thruster pushes the lander to the *left* and vice versa! Think about what you should do if both *left* and *right* keys are pressed at the same time. Test it by flying around!

Add horizontal thrusters to `MoonLander`.

15.10 Landing pad: visuals

The purpose of the game is to land on a landing pad. A landing pad is just a rectangle with some additional methods and properties. So it stands to reason to make it a descendant of the `visual.Rect` class, unfortunately, for some technical reason I have not figured out yet, this does not work for shape classes like `Rect` or `Circle`.

Create a new file *landing_pad.py* and a new class `LandingPad`. In the constructor, create a rectangle and store it in attribute (you pick the name). It should be `0.5` units wide and located at the bottom of the window but at a random position within the window horizontally. Pick the fill and line colors that you like. The only other method the class needs is `draw()`.

---

[3]Pun intended.

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

Create `LandingPad` class. Use it in *code07.py*.

15.11   Computing edges of game objects

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

We do not have information about it *directly*. We have the vertical position of the lander in `self.image.pos[1]` (I assume here that the visuals attribute is called `image`) and its height in `self.image.size[1]`. From this, it is easy to compute the bottom edge (but remember that position is about the *center* of the rectangle). Accordingly, you could create a computed property `bottom`. Create computed attributes for `bottom`, `left`, and `right` of the lander class and for `top`, `left`, and `right` of the landing pad.

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

15.12   Landing

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

- The lander must be within the limits of the lander pad horizontally.
- The vertical speed must be zero or negative (otherwise, the lander flies up) but below a certain threshold that we will define as `vertical speed threshold [norm/sec]`. I set it to `0.05`.
- The absolute horizontal speed must be below a certain threshold, also defined as `horizontal speed threshold [norm/sec] : 0.05`.

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

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

```python
if lander_is_within_horizontal_limits and \
   lander_vertical_speed_is_good and \
   lander_horizontal_speed_is_good:
```

```
    ...
else:
    ...
```

Implement landing checks in *code08.py*.

15.13   More rounds

Extend the game to have more than one round after the player either landed or crashed. Remember to reset the position of the lander before each new round. You can also add a `reset()` method to the landing pad as well, randomizing it horizontal position.

Add `reset()` method to the `LandingPad` class. Add more rounds in *code09.py*.

15.14   Limited fuel

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

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

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

Add fuel and fuel gauge to `Lander` class.

15.15   Add to it!

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

Chapter 16

Space invaders: mixins and duck typing

Today we will program a classic Space Invaders game with a twist. We will be using object-oriented programming but you will learn about *mixins* and *duck typing*. Previously, you have learned how to ensure generalization — common behavior for different classes — via inheritance. E.g., when all visual stimuli inherit from the save `BaseVisualStim` class, you can be sure that they all have `size`, `ori`, and `pos` properties and that you can `draw()` them. However, this approach may not be best suited for cases when the same class must implement several different behaviours. One solution is to use *multiple inheritance*, so that a class is a descendant of several classes and, thus, get behavior from many of them. This is the canonical way of ensuring generalization but you can achieve same means differently.

16.1 Mixins

One way to infuse a *common* behavior into *different* classes is via a mixin classes. These are classes that only define a single *behavior* (i.e., a method) and nothing else. No constructor and, typically, no attributes. Thus, they are too limited to be used on their own but can be inherited from (mixed into the proper class) enabling that single behavior in the descendant.

Think about all creatures that can fly: insects, birds, bats, astronauts, etc. They do have a common ancestor but that common ancestor did not have the ability to fly. Instead, each line evolved that ability independently and all have evolutionary "cousins" that cannot fly. When programming, you can follow the same pattern of creating a common ancestor for flying and non-flying insects, then implementing ability to fly in the former. Do the same for birds, bats,

astronauts, etc. If your implementation must be very detailed and creature-specific, this might be unavoidable. However, if your ability to "fly" is very abstract and, therefore, the same for all creatures in question, you will end up writing the same code for every insect, bird, bat class. An easy copy-paste, of course, but that means you get multiple places with identical implementation, so when you need to change it, you will have to make sure that you do it in all the places (and you gonna miss some, I always do). Alternative? Mixins! You create a class `FlightAbility` that implements that common abstract "flying" and, then, you inherit from that class whenever you need a flying someone. Mix a non-flying bird with the `FlightAbility` and it can fly! Mix it with an insect: Flying insect! You may also mix in more than one ability. Again, start with a bird that just walks around (a chicken). Add a mixin `FlightAbility` and you get a flying bird (a pigeon). Take the walking bird again and mix in `SwimAbility` and you get a swimming bird (a penguin). Mix in *both* and you get a bird that can both fly and swim (a swan)!

You may not need mixins frequently but they are a powerful way of creating an isolated behavior that different classes might need without enforcing strict inheritance structure. PsychoPy is big on mixins. For example, it has `ColorMixin` that could be mixed-in to a visual class that needs to work with color, so it implements all the repetitive[1] code for translating an arbitrary color representation (string, hexadecimal code, RGB triplet, HSL triplet, single grayscale value, etc.) into the internal RGB color value. It also has `TextureMixin` for classes that use textures for drawing objects. In our *Space Invaders* game, we will use a Mixin class to mix in a "boom sound when exploded" behavior, common to both aliens and the player's ship.

16.2   Duck typing

Alternatively, you might need your object to behave in a certain way but having a proper class hierarchy is an overkill because this is only one class and you might want some but not all the functionality. The idea is to use "duck typing", which comes from saying "If it walks like a duck, and it quacks like a duck, then it must be a duck." In other words, if the only things you care about are walking and quacking, do you need it to be an actual duck[2]? Will a goose that can walk and quack the same way do? Will a *dog* that can walk and quack like a duck do? Obviously, the correct answer is "it depends" but in a lot of situations you are interested in a common behavior rather than in a common ancestor.

Duck typing is a popular method in Python. For example, `len(object)` is a canonical way to compute length of an object. That object could be a string `len("four")`, or a list `len([1, 2, 3, 4])`, a tuple `len(tuple(1, 4, 2))`, a

---

[1]boilerplate

[2]A man is hailing a taxi on a sidewalk. A car stops next to him. The guy looks at it and says "But where is the taxi sign on the roof?". The driver replies: "Do you need a taxi sign on the roof or a ride?"

dictionary `len({"A": 1, "B": 5})`, etc. The idea is that as long as a class has a concept of length (number of elements, number of characters, etc.), you should be able compute its length via `len()`. This is achieved by adding a *special method* to the class `__len__(self)` that must return an integer. This is called a "hook method", as it is never called directly but is "hooked" to the `len()` function call. Thus, whenever you write `len(object)`, it is actually translated into `object.__len()__`[3].

Another popular duck typing application is an implementation of an iterator: an object that yields one item at a time, so you can do lazy computation[4] or loop over them. You already know how to implement an iterator as a function but it also be defined as a class with two special methods: `__iter__(self)` and `__next__(self)`. The former is called when iteration starts and it should perform a necessary initialization (e.g., setting internal counter to 0, shuffling elements, etc.) and must return an *iterator object* (typically, a reference to itself). The `__next()__` method is called whenever the next item is needed: if you use iterator in the `for` loop, it is called automatically or you can call it yourself via `next(iterator_object)`. `__next()__` must return the next item or raise a StopIteration exception, signalling that it ran out of items.

Note that an iterator does not require an implementation of the `__len()__` method! How come? If you have items you can iterate over, doesn't it make sense to also know how many (length of an object)? Not necessarily! First, if you use an iterator in a `for` loop and you just want to iterate over all of the items until you run out of them (which is signaled by the StopIteration exception). Thus, their total number and, hence, `len()` method is of little interest. Accordingly, why implement a function that you do not need[5]? Second, what if your iterator is endless (in that case, it is called a *generator*)? E.g., every time `__next__()` is called it returns one random item or a random number. This way, it will never run out of items, so the question of "what is its length" is meaningless unless you take "infinity" as an answer. This lack of need for `__len__()` for iterators is the spirit of duck typing: implement only the methods you need for your duck, ignore the rest.

Below, you will practice duck typing by implementing both `__len()__` and two special iterator methods as part of our `AlienArmada` class.

16.3 Which one to use?

Now you know *three* methods to produce common behavior: proper inheritance, mixins, and duck typing. Which one should you use? Depends on what you

---

[3]Why not implement it as a method `object.len()` or, even better, as a read-only property `object.len`? Read here for the justification.

[4]Lazy computation means that you compute or get only what is necessary right now, rather than computing or getting all items in one go.

[5]Of course, if you *do* need it, you should implement it. The point is that quite often you do not.

need. If you have many classes and you can have a well defined inheritance tree, use it. However, if you have some specific behavior that you need in some classes that is hard to fit with the inheritance tree, use mixins. If you have a single class that needs very specific functionality, such as an iterator: use duck typing. Using any one of them does not preclude you using others, but be moderate, using too many different paradigms will be confusing for you and a reader.

16.4   Space Invaders

We will program a simple version of the game with a gradually descending alien armada. Your task is to capture all aliens by firing a teleport beam before one of them rams your ship or gets away. Below, you can see my version of the game.

As per usual, the plan is to move slowly in small steps to keep complexity of changes low. Here are the steps:

- Start with our usual PsychoPy boilerplate hidden in a context manager.
- Add a `Spaceship` and use mouse to move it along the bottom of the screen.
- Create an `Alien` and then the whole `AlienArmada`.
- Add `Laser`, then a `LaserGun`, so that the player can fire many laser shots by pressing left mouse key.
- Add check for hitting aliens with the laser and teleporting them off the screen.
- Use mixin to add teleport sound.
- Use cool iterators to cycle through lasers and aliens.
- Make alien armada move.
- Check for end-of-game.

Before we start, grab images that we will use for the game space-invaders.zip, created specifically for our seminar by Andrej Pastukhov, who said you absolutely must look at this link: Wie die Pixelarts meines Sohnes Andrej entstanden sind[6].

16.5   Boilerplate

Create our usual boilerplate with an external settings file via a context manager with `GameAbort` exception handling. You can reuse the code from our last game but I strongly suggest checking that you understand every step. My suggestion for the window size is 640×480 because the sprites are 32×32 but chose whatever looks good on your screen.

Create `GameContext` and `GameAbort` classes. Put your boilerplate code into *code01.py.*

---

[6]No worries, it is safe. He says, you should understand. But I have no idea because, evidently, I am a dinosaur out of touch with modern trends.

16.6 The spaceship

Create a new file for `Spaceship` class. It is remarkably boring as it just an image that moves left-right on a horizontal line. Thus, we will create it as a descendant of ImageStim class. The only method that you need to define right now is the constructor `__init__()`. It should take Window and image filename (should be stored in settings) as parameters (plus the compulsory `self`, of course) and call `super().__init__(...)` to initialize itself via the inherited constuctor. Apart from the two parameters you have, you also need to specify the initial position of the ship via `pos`. The ship should be somewhere just above the lower edge of the window. For me, vertical position of -0.9 worked quite nicely.

In the main code, create an object and draw it in the main loop.

Create `Spaceship` class in a separate file. Use it in *code02.py*.

16.7 The spaceship moves

Now, let us use the mouse to control the horizontal position of the ship and, later on, left button to shoot. First, initialize the mouse in the context manager, so that it is one its attributes (then you can use it as `ctx.mouse`). Then, in the main loop, use getPos() method to get the position of the mouse and use the *horizontal* component to alter the position of the ship. Note that you *cannot* assign just a horizontal or vertical components to the position via `.pos[0] = ...`. In my PsychoPy 2021.1.4 it does not generate an error but does not change the actual position either. Thus, you need to assign a tuple of `(new horizontal component, original vertical component)` (the ship should stay at the same vertical position throughout the game). Test moving your spaceship around. You can make mouse invisible (see documentation) to make it more immersive.

Add mouse to `GameContext` Use mouse to move ship in *code03.py*.

16.8 An alien appears

Before we create an alien armada, let us create a single alien. The class will be very basic, which is why, we will put both `Alien` and `AlienArmada` into the same file (so, think about its filename). The class itself is similar to the `Spaceship`. The only differences are that it should take (and pass) `pos` parameter (unlike the spaceship that always appears at the same location) and that the image should be randomly picked from a tuple of filenames (*alien-1.png* and *alien-2.png*, the names should be defined in the settings in an "aliens" group of settings). For testing, place a few aliens at arbitrary locations on the screen to check that an alien image is randomly picked. Do not forget to draw them in the main loop.

Create an `Alien` class Use them in *code04.py*.

16.9   Using names to indicate access restrictions

Before we create an alien armada, we need to make a quick detour to talk about naming conventions for attributes and names of classes. Unlike most other languages, Python does not have private attributes or methods, i.e., attributes or methods accessible only from inside the object. If you come from Java or C#/C++, a thought that you can call *any* method and modify *any* attribute from outside should give you chills, as it becomes impossible to predict object's behavior. Python "solves" the access problem via a "gentlemen's agreement" that methods and attributes whose names start either with `__` or `_` should be *considered* private. In other words, you should not use them directly. In case of `__`, you are also prevented from doing it in a straightforward way, as the name is mangled. In the example below, you cannot access the `__color` attribute directly:

```python
class ExampleClass:
  def __init__(self):
    self.__color = "red"

example = ExampleClass()

# raises AttributeError
print(example.__color)
#> Error in py_call_impl(callable, dots$args, dots$keywords): AttributeError: 'Example
```

But as Python does not have truly private attributes, you can still access it via its mangled name`object._<ClassName><hidden attribute name>`:

```python
example._ExampleClass__color
#> 'red'
```

However, this is a last resort sort of thing that you should use only if you absolutely must access that attribute or method and, hopefully, know what you doing.

As noted above, you can also come across attributes with a *single* leading underscore in the name, e.g. `_color`. This is a weak "internal use" indicator, so they are fully visible. However, the leading underscore does *hint* that this attribute or method should be *considered* private. So, if you see an attribute like `_color`, you should pretend that you know not of its existence and, therefore, you never read or modify it directly. Of course, this is only an *agreement*, so you can always ignore it and work with that attribute directly[7]. However, this almost certainly will break the code in unexpected and hard-to-trace ways.

---

[7]On a side note of doing crazy things that you should not: You can replace a class method without inheritance with your own at run time, this is called a monkey patch!

16.10   An alien armada appears

One alien is neither scary, nor challenging enough. We want more! For this, we will create an `AlienArmada` class with a twist. It will be responsible for creating a grid of aliens, moving them around (our chance to learn about special setter methods), allowing outside processes to iterate over them (iterator duck typing), and reporting the number of remaining aliens (length duck typing). As usual, we will do this one step at a time.

Start by creating an `AlienArmada` class, below the `Alien` class. Our initial version will the following methods

- a compulsory constructor,
- `spawn()` that creates and places individual aliens,
- `draw()` that draws aliens,

and attributes

- `win` a PsychoPy window,
- `__pos` that is a tuple with the position of the armada center within the window,
- `aliens` which is a list with alien objects.

You must also expand your settings. By now you should already have an `Aliens` settings section with alien images' filenames. To define the armada, we need also need to add

- `"Initial position"`: center of the armada, I've picked `[0, 0.5]` but you can always optimize it later on.
- `"Grid size"`: number of aliens horizontally and vertically, I've used '[7, 3].
- `"Grid step [norm]"`: distance between aliens in "norm" units, I've picked 0.2.

Constructor: we will create and place aliens in a separate method `spawn()`, so our constructor will be very simple. As far as parameters are concerned, you need to pass a Window for creating `Alien` objects and a dictionary with armada-specific settings. Store both of them as attributes for later use, initialize `__pos` to the `"Initial position"`, initialize `aliens` to an empty list and `spawn()` them (again, all attributes must be initialized in the constructor, even if to an empty list or `None`).

In the `spawn()` create aliens on a grid based on your `"Grid size"` and `"Grid step [norm]"` settings, so they are centered at `AlienArmada`'s position (`__pos`). All aliens go to `aliens` attribute, of course. Remember that they need a list of file names for visuals but you have it covered as you have them in the settings. It should look like this:
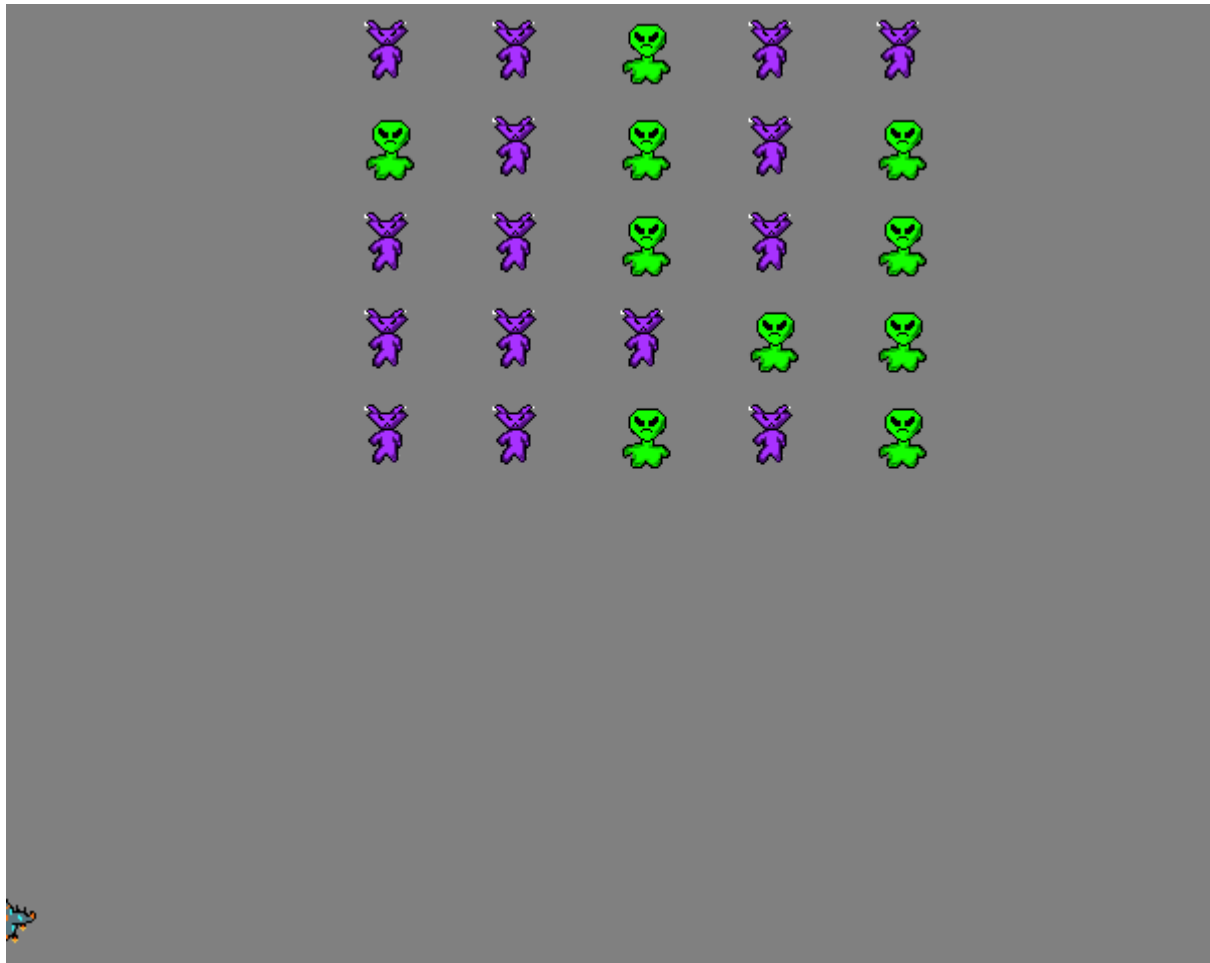
Create the `AlienArmada` class. Use it in *code05.py*.

16.11   A laser shot appears

Now we need to create `Laser` class (in a separate file). Eventually, when a player presses left mouse button, the laser should appear at the location just above the ship and fly up with a certain speed. As per usual, all parameters for laser (and `LaserCannon`) should be stored in the settings file under a separate section (e.g., `laser`). For the moment, we only need to specify `speed [norm/sec]` (laser shot speed in norm units per second, I've picked 1 norm / second, so it takes 2 seconds for the shot to fly across the screen)) and, optionally, filename of the laser bitmap (but that could be also hardcoded, as we have only one image).

Let us take care of the first step: creating a laser shot just above the sheep. Create `Laser` class as a descendant of ImageStim (just like our `Spaceship` and

`Alien` classes). It should take `win`, `pos`, and `speed` as arguments and call the ancestor constructor. The `pos` argument is the position of the ship (that fired it) but the laser must appear *above* it, so you need to adjust `pos[1]` for that (hint, you can get window height from `win.size[1]` and you know that sprites are 32 pixels high). The `speed` is in *norm units per second* from the settings. You need to save the `speed` value in an attribute for later use.

For testing, create a laser object right after you create the ship itself and draw it in the main loop. It won't move but you will be able to see whether you've got the height right.

Create the `Laser` class. Use in *code06.py.*

16.12   The laser shot flies

Let us add `fly()` method to the laser. It will be very simple, every time it is called the laser should move upwards. As in a Moonlander game, we need a clock to measure the time elapsed between the updates, so create it as an attribute in the constructor. In the `fly()` method, you need to figure out the time elapsed since the last call or since the shot was created (don't forget to reset the timer after that) and move the shot based on its `speed` and elapsed time.

Remember to call `fly()` method in the main loop and see how the laser flies up and off the screen.

Add `fly()` method to `Laser` class. Use it in *code07.py.*

16.13   We want more lasers!

Now we will create a `LaserGun` class that will take care of individual laser shots. It will 1) create a new `Laser` object whenever we `fire()` from it, 2) ensure that it does not overheat by allowing only one shot every N seconds (another parameter to be defined in the settings file, let's say 0.3 s), 3) call `draw()` and `fly()` methods for all laser shots. We will create the class `LaserGun` in the same file as the `Laser`.

In the constructor, pass PsychoPy window variable and a dictionary with laser gun specific settings (these should include the `speed` that you specified earlier and the cool down period in seconds) and store both in attributes for later use. Initialize `lasershots` attribute with empty list, and create a cool down timer.

The `fire()` method should take position of the ship as a single parameter. If enough time passed since the last shot (check the cool down timer for that), create a new `Laser` at the supplied position, add it to the `lasershots` list, and reset the cool down timer. In `draw()`/`fly()` simply draw/fly all `lasershots`.

For testing, use `LaserGun` in place of the `Laser`. Both have same `draw()` and `fly()` methods that you should be calling already. In the main loop, check if left mouse button is pressed. If it is, `fire()` the laser gun, passing current position of the ship to it.

Create `LaserGun` class. Test it in *code08.py*.

16.14   Too many lasers!

We do not need our lasers once they fly of the screen, so we should remove them. First, we need to know which `Laser` shot has `expired`": is off the screen (this section) or hit its target (the next section). For this, modify the `Laser` class by adding an `expired` attribute that is set initially to `False` (the shot is good). In addition, write a new method `check_screen_limits()` that would set `expired = True` if the shot is above the upper edge of the window but leaves `expired` as is (unmodified!) otherwise (the shot could be within the screen and get `expired` because it hit an alien).

Next, add a `cleanup()` method to the `LaserGun` class that will call `check_screen_limits()` for all shots and then remove all `expired` ones. This is both simple and tricky! Conceptually simple: you loop over `lasershots` and delete any object that is `expired`. The easy way is to do it via list comprehension (deleting objects by not including them in the updated list) but for didactic reasons we will use del instead. If you have a list and you want to delete a second element, you write

```
x = [1, 2, 3, 4, 5]
del x[1]
print(x)
#> [1, 3, 4, 5]
```

However, there is a catch. Imagine that you want to delete second and forth elements, so that the result should be `[1, 3, 5]`. If you just delete second and then forth elements, you won't get what you want:

```
x = [1, 2, 3, 4, 5]
del x[1]
del x[3]
print(x)
#> [1, 3, 4]
```

Do you see why? Solution: start deleting from the end, this way indexes of earlier elements won't be affected:

```
x = [1, 2, 3, 4, 5]
del x[3]
del x[1]
print(x)
```

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

Note that you must use `del list[index]` format, so you need to use indexing in the for loop of the `cleanup()` method:

```
for ishot in backwards-index-built-via-the-range-function:
  check if lasershot has left the screen
  if self.lasershots[ishot] needs to be deleted:
    delete that self.lasershots[ishot]
```

Add `cleanup()` call to the main loop right after the `fly()` to automatically delete expired shots. Once you fired a few shots and they are all off the screen, put a breakpoint and check that the `.lasershots` is empty (your `cleanup()` works as it should).

Add `expired` property to `Laser`. Add `cleanup()` method to `LaserShot`. Use `cleanup()` in *code09.py*.

16.15   I am hit! I am hit!

Now we need to check whether a lasershot hit an alien. In that case, all the relevant aliens and shots must be removed. Removing lasershots is easy: We just set them as `expired` and `cleanup()` does the rest. Clearly, we need the same mechanism for the aliens.

Add `hit` attribute to the `Alien` class and initialize to `False` (an alien is not initially hit). In `AlienArmada` class, implement a `cleanup()` method, analogous to one in `LaserGun` that will delete any alien that is `hit`. Add the call to it in the main loop at the same location as for the `LaserGun`. You can test it by setting `hit` to `True` for one of the aliens and it should be missing.

Now for the actual check. You should loop over all aliens and laser shots attributes of `AlienArmada` and `LaserGun` checking *every pair* on whether they overlap. If they do, set laser shot to `expired` and the alien as `hit`. Do this check *before* the clean up but *after* the laser shots fly.

Test it!

Update `Alien` and `AlienArmada` classes Check for laser shots hitting aliens in *code10.py*, .

16.16   Duck typiing iterators

In the precious exercise, you used `aliens` and `lasershots` attributes directly in the loop. However, let us inject some coolness into our code and turn `AlienArmada` and `LaserGun` classes into iterators. Recall that you need to implement two special methods for this: `__iter__(self)` and `__next__(self)__`.

The first one, initializes the loop, the second one yields the next item. The actual implementation is very simple. You need to create a new attribute that will be used to track which item you need to yield, call it `iter` and initialize it to `None` in the constructor.

In the `__iter__(self)`, initialize the counter to `0` and return `self`: a returning the reference to the iterator, which is the object itself[8]. In the `__next__(self)`, check if `iter` is within the (`aliens` or `lasershots`) list length. If it is not (i.e., you ran out aliens/shots to iterate over), `raise StopIteration`. If it is, increase the iteration counter and return the element it was indexing *before* that: Think about why you need to return `self.aliens[self.iter - 1]` rather than `self.aliens[self.iter]` or how you can use a different starting value and range check to avoid this.

Do this for both classes and use them in for loops directly, i.e. looping over `aliens` and `laser` instead of `aliens.aliens` and `laser.lasershots`.

Test it! It should work as before, of course, but with cooler duck typing inside!

Turn `LaserGun` and `AlienArmada` into iterators. Use them as iterators in *code11.py*.

16.17   Got'em all!

Now, implement `__len__()` method for `AlienArmada` class that returns the number of remaining aliens. In the main loop, use it via `len()` function to check whether you won: no aliens left, the game is over. Use `len()` with the object, not with its `aliens` list attribute!

Add `__len__()` special method to `AlienArmada` class. Use it to check whether all aliens are hit in *code12.py*.

16.18   Ping!

Now let us use a `HitSoundMixin` class that defines a single `hit()` method (no constructor!) that plays the *teleport.wav* sound[9] from materials. It can be a single line method when you both create and play the sound in one go, no temporary variable or an attribute are required. When you use it as an ancestor for the `Alien` class, it has that `hit()` method. Call it when you set *hit* to `True`.

Create `HitSoundMixin` class. Mix it into the `Alien` class. Use it in *code11.py*.

---

[8]You can cheat and return the reference to the list attribute itself and it'll do the rest but do not do it for didactic purposes.

[9]Created by Sergenious and obtained from freesound.org

16.19 Virtual attributes via getters and setters

Our next detour is to learn how you can create a private attribute (within the limits of Python, of course). Recall that we have a `__pos` attribute of an `AlienArmada` class that defines its location. The catch is that changing the position of the armada means changing position of all the aliens that constitute the armada. This means that we need write a method that receive a new position for the armada and updates both the `__pos` attribute and positions of individual aliens. And that is exactly what getter/setter methods are for. They are declared via special *decorators*[10]: `@property` and `@<name>.setter`. You already know the former one that we used to compute a value on-the-fly. But it also be used to get a value of a "private" attribute, which is why it is called a "getter" method. The latter decorator, is for a "setter" method that *sets* a new value to a private attribute. The idea of using them together is to isolate an actual attribute value from the outside influence. It is particularly helpful, if you need to control whether a new value of an attribute is a valid one, needs to converted, additional processing, etc. For example, `color` attribute of the rectangle stimulus uses this approach, which is why it can take RGB triplets, hexadecimal codes, or plain color names as a value and set the color correctly.

Here is a sketch of how it could work for a virtual attribute for `color`. One typically creates an *internal* attribute with almost the same name, e.g., `_color` or `__color`. The value is stored and read from that internal attribute by getter and setter methods:

```python
class ExampleClass:

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

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

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

---

[10]These are functions that "decorate" you function and are called *before* your function is called. They are like gatekeepers or face control, so they can alter whether or how your function is executed.

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

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


example = ExampleClass()

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

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

Note that there is no *actual* attribute `color`[11], yet, our code works as if it exists.

There is another twist to the story. If you only define the getter `@property` method but no setter method, your hidden property is read-only[12]!

16.20   The alien armada jumps

Currently, our aliens are sitting ducks. No fun! They need to get a move on. But before we can move alien armada around, we need to be able to change its position. So let us write getter/setter methods for it. To start with, define a getter, as it only needs to return the value of the hidden position attribute.

Now to more complicated and, hence, fun bit! Remember, we need to move *all* individual aliens relative to the center of the armada. For this,

1. compute the *change* in its position based on current and new value,
2. use it to alter position of individual aliens based on their current position,
3. store the new position in the private attribute.

Test it by making armada jump to a random location every time you press *space* button. The armada should keep its formation!

Implement getter/setter methods for `__pos` in `AlienArmada`. Test it in *code12.py*.

## 16.21   The alien armada moves

Our alien armada will move downwards along a sine trajectory:

---

[11] Just like there is no physical phenomenon called "color"!

[12] Note that you cannot have write-only property, you must have either getter alone or both.

$$x = x_{max} \cdot sin(2\pi f \cdot (y - y_{origin}))$$

where x_{max} is maximal deviation of the *center* of the armada from the middle of the screen (I've set it to 0.25), the f is the frequency, i.e., how fast is horizontal movement (I've set it to 3 but you can see how much slower it would be for 1 in the plot below), y is the current vertical coordinate of the armada and y_{origin} is the initial one (0.5 in my case). As with all other parameters, these should be part of the settings file plus `vertcal speed [norm/sec]` that defines how fast the armada will fly down (I would set it to 0.1, so it takes 15 seconds to fly down).
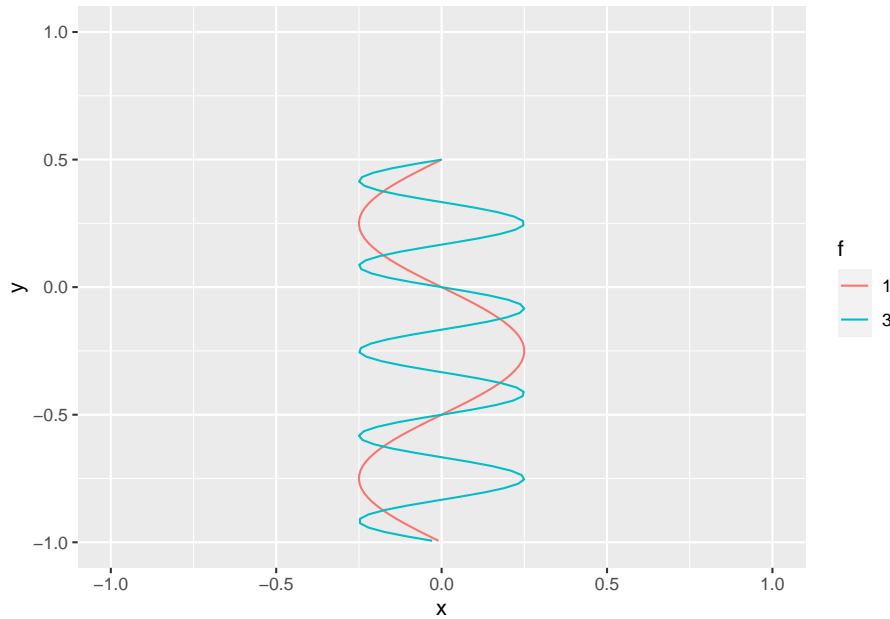
For the armada to fly, we need to add a timer (in the constructor) and define a `fly()` method that

1. computes the times since the armada creation
2. computes current vertical position based on the starting location, elapsed time (from the timer), vertical speed defined in settings
3. computes current horizontal position given the formula above and the relevant settings
4. updates current position via a simple `self.pos = (new_x, new_y)` (the setter method will take care of the rest)

In the main loop, call `fly()` when you call it for laser gun.

Add `fly()` method to `AlienArmada` class. Use it in *code13.py*.

16.22   The alien armada wins: crash!

So far, the player always wins. Let us make it more dangerous! They will lose if either an alien crashes into the spaceship or if the armada goes past the ship. Implement the former the same way as you checked whether laser shot hit an alien: loop over aliens and see if they overlap with the spaceship (game over if they do). Implement and test.

Implement crash-into-ship in *code14.py*.

16.23   The alien armada wins: missed them!

The second way to lose the game, is if at least one alien got past the ship. For this, we need to check the position of the *lowest* alien and if it is lower than that of the spaceship, the game is over. You can do it in the same loop you are checking for the hit but, for didactic reasons, let us practice `@proprety` a bit more. Implement a new computed (read-only) property `lowest_y` of the `AlienArmada` class that will return the y-coordinate of the lowest alien in the armada. You need to loop over individual aliens to find the lowest y and return it. There are different ways to do it, come up with one yourself! In the main code, add the check and make sure the game is over, if aliens got past the player.

Add `lowest_y` computed property to `AlienArmada` class. Use it in *code15.py*.

16.24   Mixin teleport

Mix in the teleport sound method to the `Spaceship` and play it if alien crashed into it.

Mix in teleport to `Spaceship` class. Use it in *code16.py*.

16.25   Game over message

Create a blinking game over message that will reflect the outcome. Something like this but use your imagination:

- "Congrats!", if the player won.
- "Crash! Boom! Bang!", if the player was hit.
- "They got away!", if the armada got past the player.

One way to simplify you life is to rename `gameover` variable into the `gamestate` variable. Initially, it could be `"playing"`, so your loop repeats `while gamestate == "playing"`. Then, you can set it based on the type of the event (`"victory"` when player wins, `"hit"` when they were hit by the armada, `"missed"` if aliens got away) and use it after the main loop to determine which message to show. Better still, use dictionary for this instead of if-else. Or "cheat" and store the message in the variable itself.

Put your updated code into *code17.py*.

16.26   We want more!

You have a solid game but it can be improved in many ways. Score? Difficulty? More levels? Aliens shooting back at you? Anything is possible!

Chapter 17

Game of life (NumPy)

Today you will be programming the Game of Life created by the British mathematician John Horton Conway. This is not really a game but rather a simulation where you set initial conditions up and observe how the population evolves. The rules, as described on Wikipedia, are very simple:
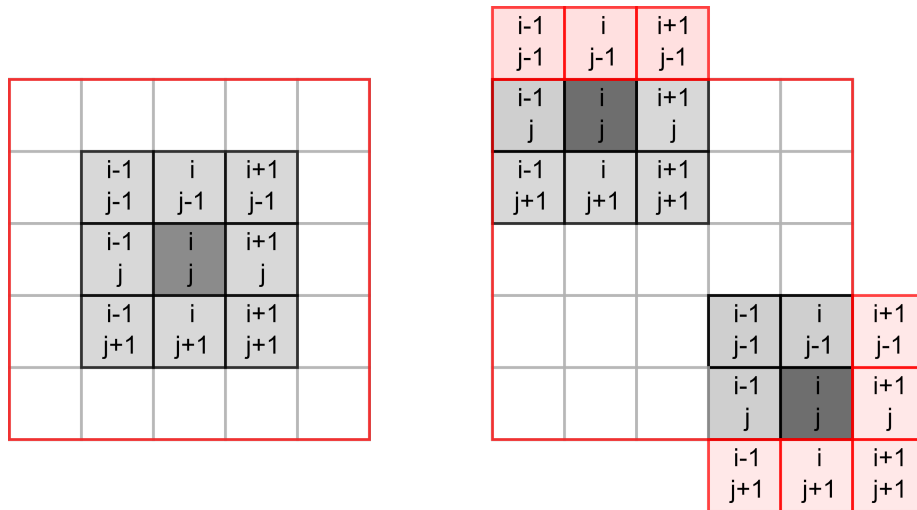
1. Any live cell with two or three live neighbors survives.
2. Any dead cell with three live neighbors becomes a live cell.
3. All other live cells die in the next generation. Similarly, all other dead cells stay dead.

Despite rules simplicity they produce remarkable evolution of a cell culture and give rise to many dynamic or static configurations. Here is an example simulation with a random starting state.

17.1   Implementing the rules

To program the game, you need to count live neighbors for every cell and use the rules above to decide whether it will be alive or dead on the *next* iteration. The latter bit is important, you need to evaluate all cells at their current state, you cannot change their status during that period. Accordingly, on each iteration you need to a create *new* cell culture and then overwrite the original one.

Our cells will live in a rectangular world, so you need to count number of neighbors in the 8 cells around it, as shown in the left hand-side image below (index / location of the dot in question is assumed to be i, j).

	i-1 j-1	i j-1	i+1 j-1	
	i-1 j	i j	i+1 j	
	i-1 j+1	i j+1	i+1 j+1	

i-1 j-1	i j-1	i+1 j-1		
i-1 j	i j	i+1 j		
i-1 j+1	i j+1	i+1 j+1		
		i-1 j-1	i j-1	i+1 j-1
		i-1 j	i j	i+1 j
		i-1 j+1	i j+1	i+1 j+1

However, things become slightly trickier, if a cell lies on the border of the world, as depicted on the right hand-side. The same rules for living or dying still apply but when counting neighbors you need to make sure that we do not try to access cells *outside* of the world (marked in red). If we mess up our indexing, at best we will get an error, at worst we will access wrong elements. E.g., when our index is negative, Python will not issue a mistake but will simply count elements *from the end.*

17.2   NumPy

Our cells live on a 2D grid which we can implement it as a list of lists. We will start with this implementation but you will see that using a matrix (a 2D array) is more natural. It also gives us opportunity to learn about NumPy that provides a powerful and fast implementation of multidimensional arrays and numerous mathematical tools. It is one of the core non-standard (i.e., not included into the base Python distribution) libraries that is so prominent that you are expected to use it via its alias rather than by its full name:

```
import numpy as np
```

I will not try to properly cover NumPy here, it is simply too vast. Rather, the purpose is to make you aware of opportunities that NumPy provides. The material below will contain just the bare minimum that we need for the game. I suggest that you read the quickstart guide to NumPy before continuing and you should be ready to consult the absolute basics for beginners whenever you have questions about manipulating arrays.

17.3 Boilerplate

Implement usual PsychoPy boilerplate with a external settings and context manager. The context manager should initialize PsychoPy window and a mouse, whic we will use it later to draw or erase individual cells. The size of the window that we need depends on the grid size that cells live at and the size of the individual cells. Define all three parameters - `grid width [cells]`, `grid height [cells]`, and `cell size [pix]` - in the settings file and compute window size on the fly. I've picked a 30 by 30 grid with each cell measuring 10 pixels (both widht and height, as cells are square).

Create `GameContext` and `GameAbort` classes. Put your boilerplate code into *code01.py*.

17.4 Visualizing an array

We will use a 2D Numpy array to encode our cells: 1 will mean a living cell and 0 will mean an empty space. But before we create the array, we need to think about how we will visualize it. There are at least three ways you can do it in PsychoPy. First, you can create a Rect (or some over visual primitive or even an image) for each cell and draw them one-by-one in a loop. However, that will be extremely inefficient and slow. Second, you can do effectively the same one-primitive-per-cell but via an efficiently implemented ElementArrayStim class. Finally, you can use a cool feature of ImageStim class. So far, we only used names of external files as its `image` input but it can also take a 2D NumPy array and display it as an image. We will use the latter approach, as it is the simplest.

Let us start by 1) creating a 2D grid, 2) converting it to a 3D image (for an RGB color space, PsychoPy expects one 2D grid of values per channel), and 3) displaying it via ImageStim.

17.5 A random matrix

There are different ways to create a matrices (2D arrays) in NumPy, see Creating matrices. Here, we will generate a matrix via NumPy random generator but you could also experiment with generating an empty grid via numpy.zeros() or a fully crowded one via numpy.ones().

Consult documentation and figure out how to generate a grid of integers that range between 0 and 1 (hint, take a look at the second example that covers generating integer values) creating a matrix of the given size (hint, take a look at the third example). I suggest playing with the code in a Jupyter notebook starting with a small size of the grid (so you could easily see it in the output, as for a 3×4 grid below).

```
#> Grid shape:  (3, 4)
```

```
#> [[0 1 1 0]
#>  [1 0 0 1]
#>  [1 0 1 0]]
```

17.6   From 2D to 3D

State of our cells is represented by a 2D NumPy array of size `[width, height]` but ImageStim expects a 3D array of size `[width, height, 3]` that defines the image in the RGB or some other color space. The simplest way to do this for our game is to stack the same 2D array three times. Zeros at all channels will give us black (empty space), whereas ones for all channels (a mixture of maximal red, green, and blue) will add to white. Again, experiment in a Jupyter notebook making sure that output and the shape of the 3D array are correct.

```
#> 3D grid shape:  (3, 3, 4)
#> [[[0 1 1 0]
#>   [1 0 0 1]
#>   [1 0 1 0]]
#>
#>  [[0 1 1 0]
#>   [1 0 0 1]
#>   [1 0 1 0]]
#>
#>  [[0 1 1 0]
#>   [1 0 0 1]
#>   [1 0 1 0]]]
```

17.7   Correct values range

We are almost ready to use the 3D array as an image but our values go from 0 to 1, whereas the default RGB color space in PsychoPy expects values to go from -1 (black) to 1 (white). Thus, we need to rescale the entire array via basic algebra. The good news is that NumPy is build on the idea of vectorization of operations, i.e., you do not need to apply the same operation to one element at a time (as you would do with Python list) but to the entire array at once.

```
import numpy as np
a = np.array([1, 2, 3, 4])
a + 10
#> array([11, 12, 13, 14])
a * 10
#> array([10, 20, 30, 40])
```

As per usual, experiment in a notebook and check that values turn out as expected:

```
#> [[[-1  1  1 -1]
#>   [ 1 -1 -1  1]
#>   [ 1 -1  1 -1]]
#>
#>  [[-1  1  1 -1]
#>   [ 1 -1 -1  1]
#>   [ 1 -1  1 -1]]
#>
#>  [[-1  1  1 -1]
#>   [ 1 -1 -1  1]
#>   [ 1 -1  1 -1]]]
```

17.8 Showing an image

Finally, we are ready to create and show the image! Use the code from a Jupiter notebook to generate a random grid of size [grid width, grid height] (from the settings), stack it, and rescale it. Then, simply create an ImageStim passing the array as image and make sure that the image spans the entire window (what should it size be, given "normal" units?) and draw it in the main loop. Put a break point and use a debug console or watch tab to check whether image on the screen correspond to the original 2D grid.

Create a random grid and show it in *code02.py*.

17.9 Slicing a border problem

We have a grid-to-image conversion but we are not out of the woods yet. The problem is the cells that live next to the border. They are of no trouble now but will become one, once we start counting their neighbors. Again, recall that we need to count all the neighbors *around* each cell and for cells at the border that means trouble with indexes.

i-1 j	i j	i+1 j		
i-1 j+1	i j+1	i+1 j+1		
			i-1 j-1	i j-1
			i-1 j	i j

We could be extra careful when indexing them, but there is a simpler solution: padding the grid with zeros. This way, we can use same rules for all *inner* cells and never check or modify the empty padding.

0	0	0	0	0	0	0
0						0
0						0
0						0
0				i-1 j-1	i j-1	i+1 j-1
0				i-1 j	i j	i+1 j
0	0	0	0	i-1 j+1	i j+1	i+1 j+1

This solution still requires extra work. Namely, we need to generate an empty 2D grid that is 2 elements wide and taller than we need, fill the inner part with random 0 and 1, and then use only that inner part when stacking a 3D array for an image. When using Python lists, we would need to use nested loops but NumPy's indexing and slicing makes it very easy.

Here is how it works for a 3×3 array.

```
a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
a
#> array([[1, 2, 3],
#>        [4, 5, 6],
#>        [7, 8, 9]])
```

The first index (called axis in NumPy) is for row and the second is for columns. You can get the second row by specifying its index alone (if no other indexes are given, NumPy assumes you want all elements along other axes).

```
a[1]
#> array([4, 5, 6])
```

Or, you can be more explicit via :, as : alone means "all elements along this axis"

```
a[1, :]
#> array([4, 5, 6])
```

Similarly, you can extract the entire *last* column (negative indexes work the same way as for Python lists) but notice that you need to explicitly mention the row axis in this case:

```
a[:, -1]
#> array([3, 6, 9])
```

Even better, you can easily extract a rectangular part of a 2D matrix by specifying both rows and columns. Note that omitting either a start or stop value means, respectively, from the first or till the last element.

```
a[:2, -2:]
#> array([[2, 3],
#>        [5, 6]])
```

Finally, this can be used not only to extract values from an array but also to modify them:

```
a[-2:, 1:] = np.zeros((2, 2))
a
#> array([[1, 2, 3],
#>        [4, 0, 0],
#>        [7, 0, 0]])
```

The information above is enough to get you going but NumPy's documentation on indexing is definitely worth the read, if you need to use it frequently.

17.10   Padding with zeros

Now that you know how to slice a 2D array, let us redo the sequence:

1. generate grid full of zeros that is two elements wide and taller than we need.
2. Fill the inner part (excluding that extra padding) with random 0 and 1.
3. Stack and rescale only the inner part for a 3D RGB array we need to an PsychoPy image.
4. Create and display the image.

Generate a zero-padded random grid and show it in *code03.py*.

17.11   Cell culture

Now that you have all the required pieces, wrap it up nicely in a `CellCulture` class. As we are displaying cells as an image, it makes sense to make it a descendant of the ImageStim. In the constructor, you only need the PsychoPy window and a dictionary with settings (so you know the grid size). You could more or less copy-paste code from the *code03.py* into the constructor and use a parent constructor to initialize it once you generated a random grid. However, we need to perform *1) extract inner part, 2) stack to 3D, 3) assign to image* on every iteration (after we updated the cell culture), so it makes sense to spin it off as a separate method (e.g., `set_image_from_array()`). Thus, you can generate the zero-padded grid, initialize parent *without* an image argument (PsychoPy allows for that) but only with window and size, and then call `set_image_from_array()`.

Replace your code in the main script with a new class. It should work as before.

Create `CellCulture` class Use it in *code04.py*.

17.12 Counting neigbors, single cell nested-loops edition

Recall the rules, we need to count live neighbors for each cell and this is how we decide on whether it lives (or gets born) or dies (or the location remains empty). Let us implement this for a single cell first via nested loops and then via slicing. I suggest that you experiment and debug in a Jupiter notebook and we will transfer the code into the `CellCulture` class once it is ready. I will illustrate this using a zero-padded 5×5 grid.

```
rng = np.random.default_rng(1423)
grid = np.zeros((7, 7))
grid[1:-1, 1:-1] = rng.integers(low=0, high=2, size=(5, 5))
grid.astype(int)
#> array([[0, 0, 0, 0, 0, 0, 0],
#>        [0, 0, 1, 1, 0, 1, 0],
#>        [0, 0, 0, 1, 1, 0, 0],
#>        [0, 1, 0, 0, 1, 1, 0],
#>        [0, 1, 1, 1, 0, 0, 0],
#>        [0, 1, 0, 0, 1, 1, 0],
#>        [0, 0, 0, 0, 0, 0, 0]])
```

Let us count the neighbors for a cell at location $(2, 2)$ (so, *third* row and *third* column). For the example grid that I generated using 1423 as a seed for the random generator, that should be 4.

0	0	0	0	0	0	0
0	0	1	1	0	1	0
0	0	0	1	1	0	0
0	1	0	0	1	1	0
0	1	1	1	0	0	0
0	1	0	0	1	1	0
0	0	0	0	0	0	0

Assume that the cell is identified via `icell_row` and `icell_col` and use nested

loops to count the neighbors (think about the values *relative* indexes you need to loop through). Note that the cell itself *does not count.* You can either use an `if-else` for this but an alternative solution would be to count everybody and *then* subtract the value of the cell itself. Implement the code and try it out with different `icell_row` and `icell_col`. Specifically test it for live cells, e.g., for $(2, 3)$ you should get four neighbors not five.

17.13   Slicing neigbors of a single cell

Nested loops are fun but slicing powers of NumPy make everything even easier. To count the neighbors, we need to extract a rectangular part of the matrix, very much the same way that we did when we dropped zero-padding. The only difference is that the rectangular part is defined relative to each cell. One the one hand, you can easily generate slicing on the fly relative to `icell_row` and `icell_col`:   `grid[(compute start row):(compute end row), (compute start col):(compute end col)]`. Just start with cells in the row and column *before* the current one and end with one *after* the current one. Make sure that you put brackets around expression for start and stop values, this is not necessary in Python[1] but it will make is easier to read them. Also, remember that start index is inclusive but stop index is exclusive (just as in the rest of Python). Here is a nice array that I have extracted:

```
#> [[0 1 1]
#>  [0 0 1]
#>  [1 0 0]]
```

For didactic purposes, let us learn a different way to build relative index via numpy.meshgrid(). It allows you to create two matrices so that corresponding cell cover all possible combinations:

```
a, b = np.meshgrid([1, 2], [3, 4])
print("a\n", a)
#> a
#>  [[1 2]
#>  [1 2]]
print("\nb\n", b)
#>
#> b
#>  [[3 3]
#>  [4 4]]
```

Use this function to create 3×3 grids of *relative* indexes for rows and columns (I called it `irel_rows` and `irel_cols`). You can hard code relative indexes or (better) use numpy.arange() to generate the sequence. and then compute *absolute* index using `icell_row` and `icell_col`.

---

[1] But *is* necessary in R, which is a common source of errors where.

```
#> [[0 0 1]
#>  [1 0 0]
#>  [1 1 0]]
```

We sliced the neighbors out but now we need to *count* them via a numpy.sum()
or, equivalently via a sum() method of an array. I tend to use the function but
this is purely a matter of taste (or, more likely, habit), so pick the one you like.

```
print(np.sum(grid))
#> 14.0
print(grid.sum())
#> 14.0
```

For our example subset, we get

```
#> Neighbor count for the example cell:  4.0
```

17.14   Live and let die

Now that you have the code to count the neighbors, let us count them for all the
inner cells. Create a *new* zeros matrix that will hold neighbor counts (same size
as the grid itself) and loop through every inner cell, computing its number of
neighbors and storing it in the new matrix. Here is the answer for the random
grid above. Use my code above to generate it, so you can check whether you
get the same answer but it is also a good idea to just count it yourself for a few
cells to check that the answer is indeed correct.

```
#> [[0 0 0 0 0 0 0]
#>  [0 1 2 3 4 1 0]
#>  [0 2 4 4 5 4 0]
#>  [0 2 5 5 4 2 0]
#>  [0 3 4 3 5 4 0]
#>  [0 2 4 3 2 1 0]
#>  [0 0 0 0 0 0 0]]
```

We have information about whether a cell is alive or dead in our `grid` matrix and
information about its neighbor count is `neighbors`. Now we can use comparison
and logical operation to implement the rules. Comparisons are easy, they work
the same way but are applied to the entire array:

```
a = np.arange(-3, 4)
print(a)
#> [-3 -2 -1  0  1  2  3]
print(a < 0)
#> [ True  True  True False False False False]
```

The logical `and` and `or` are replaced with & and | but work the same way. The
only thing to keep in mind is that comparisons must be wrapped in brackets
(otherwise you will get an error)

```
a = np.arange(-3, 4)
b = np.array([1, 0, 1, 0, 1, 0, 1])

print((a < 0 ) & (b == 1))
#> [ True False  True False False False False]
print((a < 0 ) | (b == 1))
#> [ True  True  True False  True False  True]
```

We want ones and zeros, so integers rather than logical values. We can easily convert the entire array via .astype() method:

```
(a < 0)
#> array([ True,  True,  True, False, False, False, False])
(a < 0).astype(int)
#> array([1, 1, 1, 0, 0, 0, 0])
```

Now you have everything you need to implement the first rule: cell must be alive *and* have 2 *or* 3 neighbors. Here is the output matrix:

```
#> [[0 0 0 0 0 0 0]
#>  [0 0 1 1 0 0 0]
#>  [0 0 0 0 0 0 0]
#>  [0 1 0 0 0 1 0]
#>  [0 1 0 1 0 0 0]
#>  [0 1 0 0 1 0 0]
#>  [0 0 0 0 0 0 0]]
```

Rule number two: cell must be dead *and* have 3 neighbors.

```
#> [[0 0 0 0 0 0 0]
#>  [0 0 0 0 0 0 0]
#>  [0 0 0 0 0 0 0]
#>  [0 0 0 0 0 0 0]
#>  [0 0 0 0 0 0 0]
#>  [0 0 0 1 0 0 0]
#>  [0 0 0 0 0 0 0]]
```

Finally, cell get to live because either rule #1 *or* rule #2 worked for them:

```
#> [[0 0 0 0 0 0 0]
#>  [0 0 1 1 0 0 0]
#>  [0 0 0 0 0 0 0]
#>  [0 1 0 0 0 1 0]
#>  [0 1 0 1 0 0 0]
#>  [0 1 0 1 1 0 0]
#>  [0 0 0 0 0 0 0]]
```

This final output is the new cell culture that you can assign back to the grid.

Implement all the steps carefully and double[2] check that all numbers match. Try different random grids and go through them by hand to make sure everything is correct (this is the boring but vital part of programming, I am afraid).

17.15   Update cell culture

We have covered a lot of ground but now you have *everything* to implement an `update()` method of `CellCulture` method:

1. Count the neighbors.
2. Apply rules to generate new culture.
3. Put the new culture into the `grid` attribute (or however you called it).
4. Transfer the grid to the image (you have a method for this).

In the main code, call update method on every frame and see the cells evolve!

Add `update` method to `CellCulture` class Use it in *code05.py*.

17.16   Pause

In the main loop, you hopefully check for an *escape* key press to end the game (if not, you should!) Let's also add a check for a press of *space* button that will "pause" or "continue" the simulation. I.e., space should toggle (invert) a state of a logical variable (you name it) that determines whether the `update()` method is called.

Implement pause/continue via *space* in *code06.py*.

17.17   Unleash your creativity

Random cultures are fun but, perhaps, you have an idea to try out (e.g., see a Wikipedia article on example patterns such as blinkers and gliders). Let us draw with the mouse (this is why I insisted on having it as an attribute of the context), with the left button click (index 0) adding a living cell and the right button click (index 2) erasing it.

For this, create a new `CellCulture` method (I called it `change()`) that takes a new value (0 or 1, depending on the button) and mouse position within the window. In the method, compute the row and column index of the cell keeping in mind that 1) the image spans the entire image, 2) only the inner cells are shown and can be clicked on. Once you have both indexes, assign new value to the grid. For testing, pause the game, draw a few patterns, restart and see them go!

Add `change()` method to `CellCulture` Use mouse inputs and `change()` in *code07.py*.

---

[2]triple!

17.18   Done for today

You have learned quite a few things about NumPy and implemented a cool simulation. Next time, we will learn more about Numpy while combining it with a useful ElementArrayStim I have teased about.