Discover Python and Patterns with Game Programming

Philippe-Henri Gosselin

Contents

Preface

Python is not only an incredible programming language for learning but also an efficient solution for real applications. Thanks to all the improvements made in the past thirty years, Python has a unique programming style that leads to compelling and readable code.

Software design is also an excellent tool for development; unfortunately, it is less popular, perhaps because people often introduce it theoretically. Nevertheless, software design is a pragmatic solution to real problems. One can compare it to the organization of a restaurant, for which no one would think of obscure abstract concepts!

This book proposes to discover these programming tools through the creation of video games. Therefore, we don't list the many programming features with minimal examples as many pages do on the Internet. Instead, we run into some common issues during game development and see what Python or software design features we can use. This approach dramatically changes what the reader can learn. Most of the time, people present features but don't tell where we need them. Instead, we learn to identify the common problems in application development and what to do in each case.

We create a minimal text-based game, a small tank game with 2D graphics, and a strategy game with GUI components. The first game introduces the basics of functional programming and can be read by new programmers. The second game presents containers, object-oriented programming, and simple design issues and solutions. Finally, the third game introduces tools that expert programmers and designers use on a daily basis, such as "pythonish" code and event handling.

Who this book is for

This book targets different levels of knowledge in Python and software design, from people new to programming to developers with some basics. Consequently, more experimented programmers can skip the first chapters, while the new ones should read them carefully.

We dedicate the first chapter to new programmers: they don't need to know another programming language. Only the very first concepts are required, like understanding what a variable and addition are.

The following chapters require the skills presented in the first one: loops, branching, and functional programming. For these chapters, the readers don't need any knowledge in software design. Unfortunately, most courses poorly teach these topics, and in many cases, they wholly ignore them. As a result, we don't expect any basis and spend as much time as possible to present the first concepts.

The last chapters also assume that the tools presented in the previous ones are understood. It includes features of most programming languages, like lists, dictionaries, classes, inheritance, etc. Furthermore, Python programming style is not required. Concerning algorithms, the reader should know the basic ones, like iterating through a container. Finally, these chapters assume that the reader already saw the Game Loop and Observer patterns and can identify dependencies issues.

What this book is

As the title suggests, this book is a discovery. It is intended to be read from the beginning to the end. Throughout the chapters, we follow the development of games and discover what can happen and what we can do.

This book is not a collection of concepts one can pick in random order. If one needs to get details about programming features, web search engines do the job. Instead, the purpose of the book is to connect neurons in the brain and get skills like Python style and the mind of a software architect. Seeing concrete examples, again and again, is the only solution to achieve this goal.

About the author

Philippe-Henri Gosselin, Ph. D., is a Software Architect and a Data Scientist at Interdigital R&D. Before this position, he was a full professor at ENSEA, an engineering school on informatics and electronics. He was the head of the last-year “Informatics and Systems” specialty and designed all the courses and related content to learn advanced software engineering.

CHAPTER 1

Write your first game

This first chapter shows the basics of the Python language and a first pattern: the Game Loop pattern. These features are enough to create the first game: “guess the number”!

If you already have some programming skills (variables, conditions, loops, functions), you can skip this chapter, or at least have a look to the “What to remember” sections.

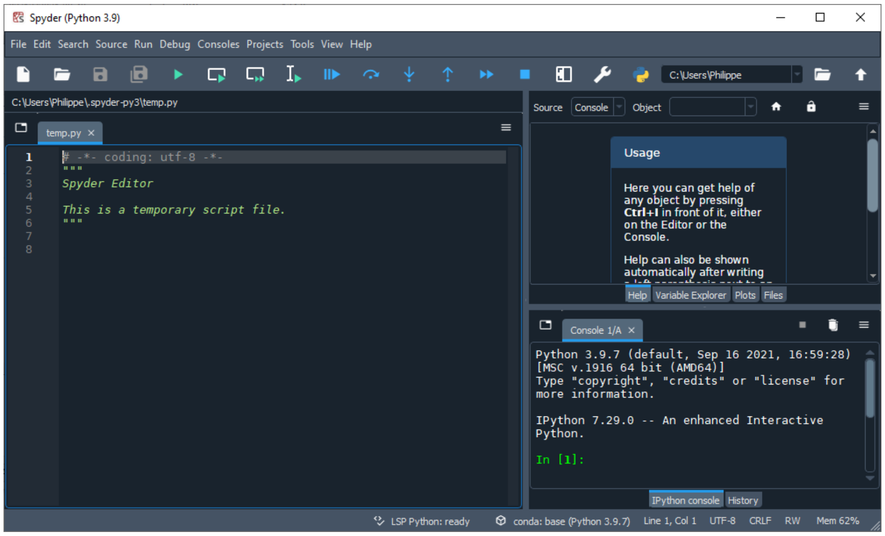
You can find examples from this chapter here: `https://github.com/philippehenri-gosselin/discoverpythonpatterns/tree/master/chap01`. The names of the examples begin with the corresponding section. For instance, “1_8_2_function.py” corresponds to Chapter 1, Section 8.2. The first section of this chapter explains how to run these examples.

1.1 Write your first program

1.1.1 Environment

Before starting anything, we need an environment to run Python programs and an integrated development environment (IDE) for code edition. This chapter uses popular ones: Anaconda for Python execution (`https://www.anaconda.com`) and Spyder for code edition (`https://www.spyder-ide.org/`). Anaconda is a tool that contains Python with many libraries and applications. Spyder is an IDE dedicated to Python, and its use is simple and easy.

To get Anaconda and Spyder, download the Python 3 distribution here: `https://www.anaconda.com/download`. The code of this book was tested with Python 3.11: if you have any issue, switch to this version. Run the installer, and be patient! Installation takes a while. At the end of the installation, you can check the boxes if this Anaconda is the main Python environment you want to use. Otherwise, don't touch anything; you can easily run programs from Spyder without these options. Then, if you are using Windows, start Spyder from the start menu (type `Spyder` in the search bar), and you should see something like this:


Spyder (Python 3.9)
File Edit Search Source Run Debug Consoles Projects Tools View Help
C:\Users\Philippe
C:\Users\Philippe\.spyder-py3\temp.py
temp.py
Source Console Object
Help Variable Explorer Plots Files
Console 1/A
IPython console History
LSP Python: ready conda: base (Python 3.9.7) Line 1, Col 1 UTF-8 CRLF RW Mem 62%

```
# -*- coding: utf-8 -*-
"""
Spyder Editor

This is a temporary script file.
"""
```

Usage

Here you can get help of any object by pressing **Ctrl+I** in front of it, either on the Editor or the Console.

Help can also be shown automatically after writing

```
Python 3.9.7 (default, Sep 16 2021, 16:59:28)
[MSC v.1916 64 bit (AMD64)]
Type "copyright", "credits" or "license" for
more information.

IPython 7.29.0 -- An enhanced Interactive
Python.

In [1]:
```

The left part is the current Python program:

```
# -*- coding: utf-8 -*-
"""
Spyder Editor

This is a temporary script file
"""
```

These lines are only comments, and they don't do anything if you run them. The first line starting with a '#' is a single-line comment, and the following are multi-line comments, beginning and ending with a triple-double quote.

The upper right part shows some help - there are other tabs we'll see later.

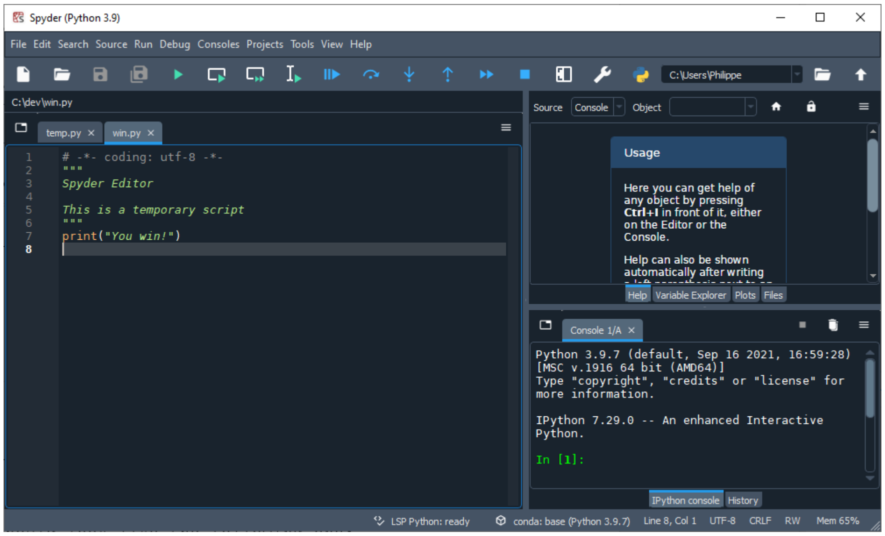
The lower right part shows the console - this is where we can see messages (from Python or our program).

1.1.2 Create your first program

Let's now create a new program from the menu **File / New File...**. A new tab appears in the top left part, with already some content: there are only comments; they don't lead to any action. Type `print("You win!")` right after, and you should get something like:

```python
# -*- coding: utf-8 -*-
"""
Spyder Editor

This is a temporary script
"""
print("You win!")
```

This first program is an easy game: nothing to do you always win :) Save it from the menu **File / Save**. Your interface should look like the following one:



```python
# -*- coding: utf-8 -*-
"""
Spyder Editor

This is a temporary script
"""
print("You win!")

```

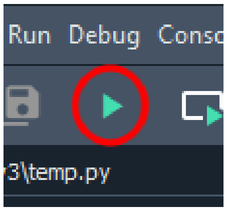
Here you can get help of any object by pressing **Ctrl+I** in front of it, either on the Editor or the Console.

Help can also be shown automatically after writing

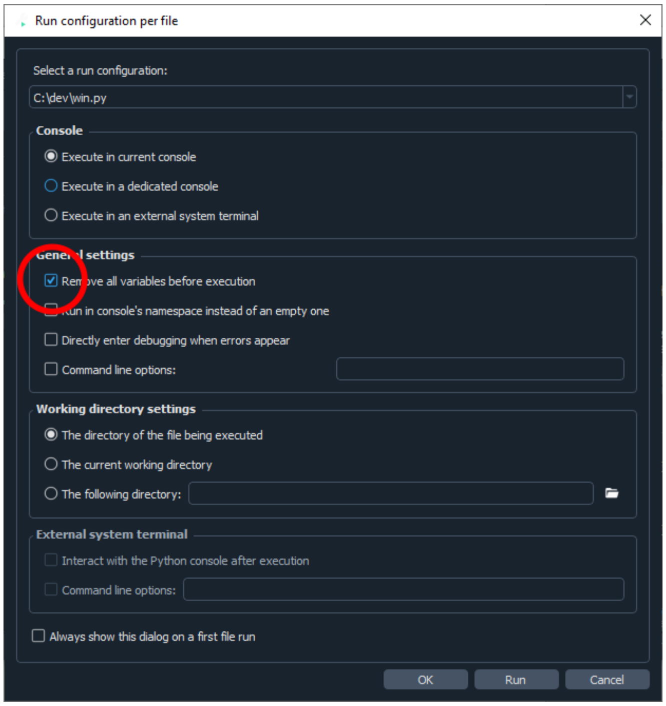
```
Python 3.9.7 (default, Sep 16 2021, 16:59:28)
[MSC v.1916 64 bit (AMD64)]
Type "copyright", "credits" or "license" for
more information.

IPython 7.29.0 -- An enhanced Interactive
Python.

In [1]:
```

1.1.3 Run your first program

To run this program, **click** the play icon in the top bar:



You should see a new dialog. **Check** "Remove all variables before execution": it will prevents unexpected errors later:



Run configuration per file

Select a run configuration:
C:\dev\win.py

Console
[✓ Execute in current console]
Execute in a dedicated console
Execute in an external system terminal

General settings
[✓ Remove all variables before execution]
Run in console's namespace instead of an empty one
Directly enter debugging when errors appear
Command line options:

Working directory settings
[✓ The directory of the file being executed]
The current working directory
The following directory:

External system terminal
Interact with the Python console after execution
Command line options:

Always show this dialog on a first file run

OK Run Cancel

Then, **click** "Run", and in the console in the bottom right part, you should see the following new lines:

```
In [1]: runfile('C:/dev/win.py', wdir='C:/dev')
You win!

In [2]:
```

The first line:

```
    In [1]: runfile('C:/dev/win.py', wdir='C:/dev')
```

is a statement Spyder typed itself to run the python program “C:\dev\win.py” in the directory “C:\dev” (Spyder uses slash “/” instead of “\”, it works the same). Do not worry about this line.

The second line

```
    You win!
```

is the result of our statement `print("You win!")`.

1.1.4 The print() function

The statement `print("You win!")` is a *function call*. We can recognize a function call when an open parenthesis follows a name. For instance, the following statement:

```
print()
```

calls the `print` function. Since there is no argument, it only writes an empty line.

Inside the parenthesis, we can add items called *arguments*. In our example, we have one argument, “You win!”. The double quotes mean that this is string content that Python should not interpret. Otherwise, if we type `print(You win !)`, the interpreter tries to execute `You win !`, but it has no meaning in Python.

We can also call functions with several arguments using commas; for instance, with the print function, we can type:

```
print("You", "Win!")
```

It prints the same message since the print function adds a space between each argument. For instance, if we run:

```
print("Wo", "rd")
```

It prints (there is a space between Wo and rd):

```
Wo rd
```

We can add more arguments, for instance:

```
print("This", "is", "my first", "game")
```

prints:

```
This is my first game
```

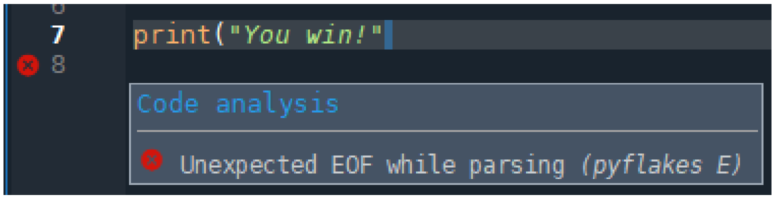
The print function can do much more; we will see its features throughout the book.

1.1.5 Syntax errors

Python, as for all programming languages, has strict syntax rules. If we make mistakes in our program, it is impossible to run it. For instance, if we forget the last parenthesis in the previous example:

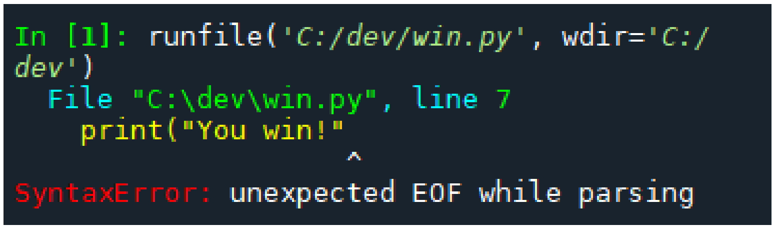
```
print("You win !"
```

Spyder shows us the error with a red icon to the left. If we leave the cursor on this icon, we get some help:



```
7    print("You win!"
8
     Code analysis
     Unexpected EOF while parsing (pyflakes E)
```

If we run this line, we see a long error message in the console (bottom right part of Spyder):



```
In [1]: runfile('C:/dev/win.py', wdir='C:/
dev')
  File "C:\dev\win.py", line 7
    print("You win!"
                    ^
SyntaxError: unexpected EOF while parsing
```

It highlights a syntax error in our program (here C:\dev\win.py) at line 8.

1.1.6 What to remember

Python environment: we need an environment to run Python programs; Anaconda is the most popular and runs on all platforms.

Integrated Development Environment (IDE): it is better to use a dedicated editor. It can run our program and help us in many ways; for instance, it can show errors without running the program. We use Spyder in this chapter: it is a good choice for beginners.

Function call: we can use many features of the Python language calling functions. The syntax is:

```
<name>(<argument1>, <argument2>, ...)
```

where `<name>` is the name of the function, and `<argument1>`, `<argument2>` are the (optional) parameters.

The print() function: displays a message in the console. Don't forget to add the double quotes around the message!

```
print("A message")
```

The `print()` function adds a space between arguments. For instance:

```
print("This", "is", "a", "message")
```

displays:

```
This is a message
```

1.2 Basic interaction

We can show messages with the `print()` function, but we need to interact with the player to create games!

1.2.1 Wait for a key

We create a new python program and type the following line:

```
input("Press Enter to continue...")
```

We run the program, and in the spyder console, we find the same line starting with "In [1]:", and the message "Press Enter to continue…":

```
In [1]: runfile('C:/dev/press_enter.py', wdir='C:/dev')

Press Enter to continue...
```

There is no line starting with "In [2]:" as before: it means that the console is waiting for user input. We click on the console window and then press **Enter** to end the program. Then, a line starting with "In [2]:" is visible:

```
In [1]: runfile('C:/dev/press_enter.py', wdir='C:/dev')

Press Enter to continue...

In [2]:
```

1.2.2 The input() function

The `input()` function can have one argument: the message to display. It also has a return value: the message typed by the user. If you run the following program:

```
name = input("Enter your name: ")
print("Your name is", name)
```

and if you type your name in the Spyder console, you get a result like:

```
In [1]: runfile('C:/dev/enter_name.py', wdir='C:/dev')

Enter your name: phylyp
Your name is phylyp

In [2]:
```

We store the return value of the `input()` function in the variable `name`. Then, we print the message "Your name is" followed by the content of the variable `name` (with one space in between).

Note that, if we type this:

```
print("Your name is","name")
```

We get the following result:

```
Your name is name
```

Since quotes surround `name`, Python does not analyze it and copy it. If we remove the double quotes (as before), Python prints the content of `name`.

1.2.3 Format strings

There are many ways to combine strings, like the one we saw with the arguments of the `print()` function. We can also use the following syntax to create the same result:

```
print("Your name is {}".format(name))
```

With this syntax, each opening/closing curly brackets “`{}`” is replaced by the value of the arguments in the following `format()` function. Note the dot `.` between the message and the `format()` function (we’ll see later what it means).

The advantage of this syntax is that spaces are not mandatory between values. For instance, if we type:

```
print("Your name is '{}'".format(name))
```

We get the following result:

```
Your name is 'phylyp'
```

With `print()` arguments:

```
print("Your name is '",name,"'")
```

we get:

```
Your name is ' phylyp '
```

Notice the spaces that surround the name.

We can repeat the curly brackets if we want to print the content of several variables. For instance, the following program:

```
firstName = "Guido"
lastName = "van Rossum"
print("{} {} is the creator of Python".format(firstName, lastName))
```

prints:

```
Guido van Rossum is the creator of Python
```

f-Strings: since Python 3.9, we can build messages with “f-Strings”. We create them with a “f” before the first double-quote:

```
firstName = "Guido"
lastName = "van Rossum"
print(f"{firstName} {lastName} is the creator of Python")
```

Python replaces each pair of curly brackets with the content of an expression; this time, it is inside the curly brackets, which leads to a more compact and readable syntax.

1.2.4 Variables and flow

We use the variable `name` to store what the user typed. We can also use variables to store other content; for instance, we can store the string combination in a variable before printing it:

```
message = "Your name is '{}'".format(name)
print(message)
```

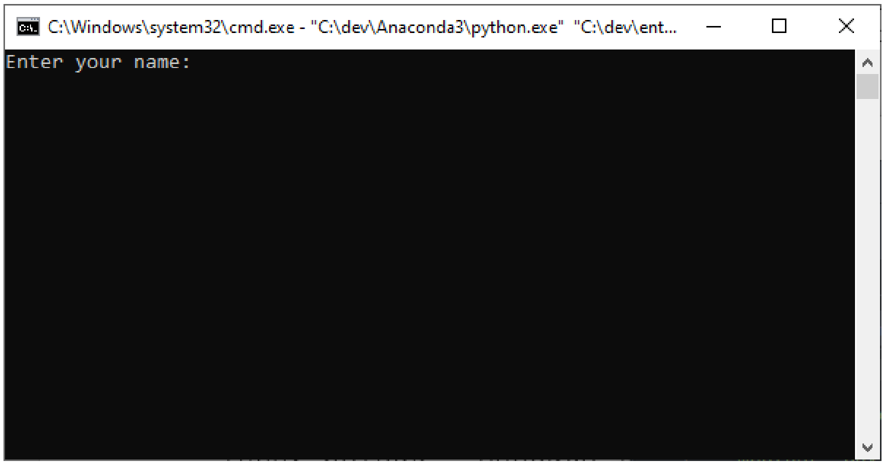
It does not change the final result (we print the same message), but it better shows how Python works internally. Even without the `message` variable, Python first computes the string combination in a variable and then calls the `print()` function with the content of this variable.

For now, remember this: Python first computes function arguments and then calls the function. If there is an error during the computation of an argument, Python never calls the function.

1.2.5 Run in a system console

You can also run Python programs in a system console. Open the run options dialogs using the menu **Run / Configuration per file...** Then, select **Execute in an external system terminal** in the console section.

If you run the program, a system console pops up:


C:\Windows\system32\cmd.exe - "C:\dev\Anaconda3\python.exe" "C:\dev\ent...

```
Enter your name:
```

You can type a name as before, but the system console disappears once you hit the Enter key. If you want to see the message "Your name...", add a third line to your program that waits for a key:

```
name = input("Enter your name: ")
print("Your name is",name)
input("Press Enter to continue...")
```

1.2.6 What to remember

The input() function: it returns what the user typed. It can also display a message if there is an argument. The usual syntax is:

```
<value> = input(<message>)
```

where <value> is the name of the variable that contains what the user typed, and <message> is the message to display.

Variables: we can store values in variables. It can be an immediate value or the result of an expression. For instance:

```
name = "Guido"
```

After this line, the `name` variable contains the string "Guido". Every time we type `name`, Python evaluates it as "Guido", until we change its value.

The following code

```
name = "Guido"
print(name)
name = "van Rossum"
print(name)
```

displays:

```
Guido
van Rossum
```

Even if lines 2 and 4 are the same (`print(name)`) the result is different because we changed the value of `name`.

Format string: we can combine messages and expressions content with the following syntax:

```
<message with curly brackets>.format(<expression1>, <expression2>)
```

Python replaces each pair of curly brackets “`{}`” in the message with the content of the arguments in the `format()` call.

Example:

```
country = "USA"
message = "{} is in the {}".format("New York", country)
```

After these lines, `message` contains the string:

```
New York is in the USA
```

With Python 3.9, we can also use f-Strings. They start with an “f” before the first double-quote and replace the expression in the curly braces with their evaluation:

```
city = "New York"
country = "USA"
message = f"{city} is in the {country}"
```

1.3 Branching

Programs in the previous section always lead to the same result. In this section, we see the basics of *branching*, or how to change the flow of a program.

1.3.1 The if statement

To display a message when the player types a specific word, we can use the following syntax:

```
word = input("Enter the magic word: ")
if word == "please":
    print("This is correct, you win!")
print("Thank you for playing this game!")
```

If you run this program and type “please”, you see the message “This is correct, you win!” as well as the last message. If you don’t, you only see the last message, “Thank you for playing this game!”.

In this example, we use the if statement:

```
if <condition>:
    <what to do if the condition is met>
<these lines are always executed>
```

If the `<condition>` is evaluated as `True`, Python executes the block that follows the `if` statement. If it is evaluated as `False`, then Python ignores it.

In the example above, the condition is `word == "please"` and we can translate it into: is it true that `word` equals “please”?

Please note the double equal symbol “==”: this is not the single equal symbol, which is the assignment. For example, if you type `word = "please"`, it means `word` takes the value “please”, and Python evaluates it as `True` (because “please” is not `None`).

Equality is an example of condition expression. Throughout the book, we will see many other cases.

1.3.2 Blocks and indentation

To understand what the `if` statement executes, you need to understand blocks in Python. In this language, indentation defines blocks: all lines with the same space symbols before the expression are in the same block.

In the example, we have a single line `print("This is correct, you win!")` with an indentation of four spaces. If we want to add a second line to this block, we must also start it with four spaces:

```
word = input("Enter the magic word: ")
if word == "please":
    print("This is correct.")
    print("You win!")
print("Thank you for playing this game!")
```

The second line can't begin with 3 or 5 spaces or a tab symbol with a width of 4 spaces. It must start with the same combination of space symbols. If we want to use, for instance, an indentation of 3 spaces, all block lines must begin with precisely three spaces. You can choose a different combination for each block.

1.3.3 Negation

In the previous example, there is no specific display if the player doesn't type the magic word. We can also use the if statement to display a message for this case:

```
word = input("Enter the magic word: ")
if word == "please":
    print("This is correct, you win!")
if not (word == "please"):
    print("This is not correct, you lost :-(")
print("Thank you for playing this game!")
```

We add a new `if` statement with the condition `not (word == "please")`. The syntax `not (<condition>)` means that we want to evaluate the opposite of a condition, in this example `word == "please"`.

We can simplify this negation the operator `!=`:

```
if word != "please":
    print("This is not correct, you lost :-(")
```

We can translate `word != "please"` into: is it true what word is different from "please"?

1.3.4 The else statement

The negation works fine, but we need to repeat the first condition. A more convenient syntax is thanks to the `else` statement:

```
word = input("Enter the magic word: ")
if word == "please":
    print("This is correct, you win!")
```

```
else:
    print("This is not correct, you lost :-(")
print("Thank you for playing this game!")
```

This program has the same behavior as the previous one: if the player types the magic word, we display the winning message, otherwise the losing message.

1.3.5 Blocks and sub-blocks

We can imagine more messages in our example depending on what the player has typed. We can use the previous syntax to do so:

```
word = input("Enter the magic word: ")
if word == "please":
    print("This is correct, you win!")
else:
    if word == "mercy":
        print("You must find the right one, you lost :-(")
    else:
        print("This is not correct, you lost :-(")
print("Thank you for playing this game!")
```

When the player types “mercy”, we display a new message. Note the block of the `else` statement:

```
if word == "mercy":
    print("You must find the right one, you lost :-(")
else:
    print("This is not correct, you lost :-(")
```

It contains four lines and two sub-blocks. The first sub-block is:

```
    print("You must find the right one, you lost :-(")
```

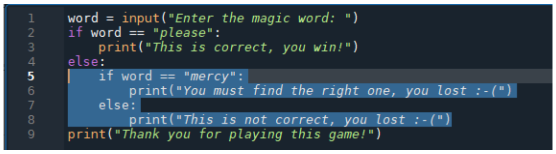
And the second sub-block is:

```
    print("This is not correct, you lost :-(")
```

We identify the blocks thanks to the indentation. For the `else` statement block, each line starts with four spaces. Then, each sub-block also starts with four spaces, to which we added four more spaces. These sub-blocks must begin with four spaces since they are part of the main block. Then, we can freely add any combination of spaces to create sub-blocks. In this example, and for most of the programs in this book, we add four more spaces, but we could also add 3 or 5 more spaces.

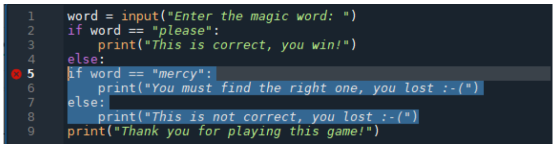
1.3.6 Indentation with Sypder

To help you understand indentation better, **select the lines** of the `else` statement in Spyder:



```
word = input("Enter the magic word: ")
if word == "please":
    print("This is correct, you win!")
else:
    if word == "mercy":
        print("You must find the right one, you lost :-(")
    else:
        print("This is not correct, you lost :-(")
print("Thank you for playing this game!")
```

Then, **hit Shift+Tab** or **select the menu Edit / Unindent**. You get the following result, where Spyder complains about the wrong indentation of the first line of the else statement:



```
word = input("Enter the magic word: ")
if word == "please":
    print("This is correct, you win!")
else:
if word == "mercy":
    print("You must find the right one, you lost :-(")
else:
    print("This is not correct, you lost :-(")
print("Thank you for playing this game!")
```

If you **hit Tab** or **select the menu Edit / Indent**, you come back to the initial state of the program.

You can play with these two operations, “Unindent” and “Indent” to see what happens when you create or delete new blocks, sub-blocks, sub-sub-blocks, etc.

1.3.7 The elif statement

The previous syntax works well but quickly becomes complex if we add more cases. We can get a better syntax with the `elif` statement:

```python
word = input("Enter the magic word: ")
if word == "please":
    print("This is correct, you win!")
elif word == "mercy":
    print("You must find the right one, you lost :-(")
else:
    print("This is not correct, you lost :-(")
print("Thank you for playing this game!")
```

After the `if` statement, we can repeat as many `elif` statements as we want. Then, we can add an `else` statement if we need to, but this is not mandatory.

1.3.8 What to remember

if/elif/else statements: allow us to change the flow of the program. With them, we can execute or ignore parts of the code. The syntax is:

```
if <condition1>:
    <what to do if <condition1> is met>
elif <condition2>:
    <what to do if <condition2> is met>
elif <condition3>:
    <what to do if <condition3> is met>
else:
    <what to do if no condition is met>
```

- Conditions are boolean expressions that evaluate as `True` or `False`;
- The `if` statement is mandatory;
- The `elif` statement is optional; we can add as many as we want between `if` and `else`;
- The `else` statement is optional; we can only have one, and it must be the last one.

Condition expressions: we know three different ways to compare values. Each of the following expressions returns `True` or `False`:

```
city == "New York"      # does city equals "New York"?
city != "New York"      # does city is different from "New York"?
not(city == "New York") # does city equals "New York" is `False`?
```

Blocks: we create them with indentation, contrary to many languages that use symbols. With an IDE like Spyder, we can indent and unindent selected lines using **Tab** and **Shift+Tab**.

1.4 Loops

The previous game only allows one try: the player must restart it to propose another word. Instead, we can repeat the game with loops until the player finds the magic word.

1.4.1 The while statement

The `while` statement allows to repeat a block:

```
while True:
    word = input("Enter the magic word: ")
    if word == "please":
        print("This is correct, you win!")
    else:
        print("This is not correct, try again!")
print("Thank you for playing this game!")
```

The `while` statement has the following syntax:

```
while <condition>:
    <what to repeat>
<these lines are not repeated>
```

It repeats the sub-block as long as the `<condition>` evaluates as `True`. This condition works the same way as in the `if` statement.

If we run this program, the game repeats forever since the condition is always `True`, and we never display the final message "Thank you. . . ". To stop its execution in Spyder, **click** the red square in the console toolbar:

To end the program when the player finds the magic word, we use a variable that states if the game is over or not:

```
1 repeat = True
2 while repeat:
3     word = input("Enter the magic word: ")
4     if word == "please":
5         print("This is correct, you win!")
6         repeat = False
7     else:
8         print("This is not correct, try again!")
9 print("Thank you for playing this game!")
```

We initialize a `repeat` variable as `True` (line 1). We repeat the block in the `while` statement as long as `repeat` is `True` (line 2). If the player finds the magic word (line 4), we print the winning message (line 5) and set `repeat` to `False`. In this case, the next time we evaluate the `while` condition, it is `False`, and the loop stops.

1.4.2 The break statement

The previous approach does not stop the loop immediately. It means that we still execute the lines following `repeat = False`, even if we want to stop the loop. We can get a better result with the `break` statement:

```
1 while True:
2     word = input("Enter the magic word: ")
3     if word == "please":
4         print("This is correct, you win!")
5         break
6     else:
7         print("This is not correct, try again!")
8 print("Thank you for playing this game!")
```

When we execute the `break` statement (line 5), the loop ends, and we go straight to line 8. Note that we no more need to introduce a variable to control the flow.

1.4.3 The continue statement

Another way to control loops is thanks to the `continue` statement:

```
1 while True:
2     word = input("Enter the magic word: ")
3     if word != "please":
4         print("This is not correct, try again!")
5         continue
6     print("This is correct, you win!")
7     break
8 print("Thank you for playing this game!")
```

The `continue` statement does not stop the loop; it goes to the beginning of the loop directly, as if the loop block was ending where `continue` is. Note that Python evaluates the `while` condition before continuing the loop.

In the example, if the player doesn't type the magic word (line 3), we display a message (line 4), and we stop the current block (line 5) and go back to the beginning of the loop (line 1).

1.4.4 Loops in loops

The `break` and `continue` statements can be in any sub-block of a loop, like an `if` or `else` block. Python looks for the first loop containing the keyword and breaks or continues. Let's see an example:

```
while True:
    while True:
```

```
3          word = input("Enter the magic word: ")
4          if word == "please":
5              print("This is correct, you win!")
6              break
7          else:
8              print("This is not correct, try again!")
9      word = input("Type 'again' to play again: ")
10     if word != "again":
11         break
12 print("Thank you for playing this game!")
```

This program runs the game as before in lines 2-8. It works the same; being in a loop does not change its behavior. The `break` statement in line 6 stops this inner loop.

The main loop in lines 1-11 repeats the game if the player types “again”. The `break` statement in line 11 stops the main loop.

If we want the sub-loop in lines 2-8 to stop the main loop, we can use a variable:

```
1  repeat = True
2  while repeat:
3      while True:
4          word = input("Enter the magic word: ")
5          if word == "please":
6              print("This is correct, you win!")
7              break
8          elif word == "quit":
9              repeat = False
10             break
11         else:
12             print("This is not correct, try again!")
13 print("Thank you for playing this game!")
```

We initialize a `repeat` variable to `True`. The main loop continues as long as it is `True` (line 2). If the player types “quit” (line 8), we set `repeat` to `False` (line 9) and stop the inner loop (line 10). Then, since `repeat` is `False`, the main loop stops, and we display the final message (line 13).

1.4.5 What to remember

The while statement: it repeats a block:

```
while <condition>:      # Repeat as long as <condition> is True
    <what to repeat>
<these lines are not repeated>
```

The continue statement: it goes straight to the beginning of the loop:

```
while <condition>:
    <what to repeat>
    ...
    continue # Go to the beginning, and repeat if <condition> is True
    <these lines are never executed>
```

The break statement: it stops the loop immediately:

```
while <condition>:
    <what to repeat>
    ...
    break  # Stop the loop
    <these lines are never executed>
```

1.5 Type basics

1.5.1 Strings

In previous programs, we used strings, for instance:

```
message = "Hello"
```

The `message` variable is a string. The name of strings in Python is *str*.

We can also use simple quotes to create strings:

```
message = 'Hello'
```

We can ask for the type of an expression with the `type()` function:

```
print(type(message))
```

If we run this line, we get the following result:

```
<class 'str'>
```

It tells us that `message` is an instance of the `str` class. We will see later what a class is.

We saw how to build strings with `format()` and f-Strings. We can also use operators, for instance:

```
print("Hello" + " world!")
```

This line displays “Hello world!”. We can get the same result with this syntax:

```
message = "Hello"
message += " world!"
print(message)
```

We can also use the multiplication operator "*" with strings to duplicate a message:

```
very = " very" * 2
print("Thank you" + very + " much!")
```

The `very` variable contains " very very" which leads to the following result:

```
Thank you very very much!
```

1.5.2 Booleans

We also saw boolean values. Python calls them *bool*. We can check that with the `type()` function:

```
b = True
print(type(b))
```

It displays:

```
<class 'bool'>
```

Their value is `True` or `False`. Conditional expressions return boolean values, using the equality operator "==", the inequality operator "!=" or the negate operator `not`.

1.5.3 Integers

With Python, we also manipulate numbers with a data type called "int" (for integer). It allows the storage and computation of natural numbers (numbers without a decimal part).

We can create a variable with an integer value in the following way:

```
magicNumber = 3
```

Python understands that `magicNumber` is an `int`: there are only digits and no dot. If we use quotes, then it becomes a `string`:

```
magicNumber = "3"
```

We can express many mathematical formulas with Python integers. Most of them are straightforward, for instance:

```
x = 2
y = 3
sum = x + y    # Sum
mul = x * y    # Multiplication
div = x // y   # Division
rem = x % y    # Division remainder
```

Note well the division operator "//" for integers. If we use a single slash "/", the division is for float numbers!

If we need to combine operations, we can use parenthesis as in mathematical formulas:

```
result1 = x + (4 * y)   # First compute 4 * y, then add x
result2 = (x + 4) * y   # First compute x + 4, then multiply by y
```

Some statements combine assignment and operation:

```
x += 4    # Add 4 to x
y *= 2    # Multiply y by 2
```

```
x //= 4   # Divide x by 4
y %= 2    # Store the remainder of y by 2 in y
```

1.5.4 Types and operators

Types change the behavior of operators. For instance, if you run the following program:

```
magicNumber = 3
print(2 * magicNumber)
```

We see the value “6”, because `magicNumber` is an `int`, and the result of 2 times 3 is 6.

If we run this other program:

```
magicNumber = "3"
print(2 * magicNumber)
```

We see the value “33”, because `magicNumber` is a `string`, and the result of 2 times “3” is “33” (Python repeats the string two times).

1.5.5 Typecasting

If we want to implement the game “guess the number” with an integer variable, we can try the following program:

```
1  magicNumber = 3
2  while True:
3      playerNumber = input("What is the magic number? ")
4      if playerNumber == magicNumber:
5          print("This is correct! You win!")
6          break
7      print("This is not the magic number. Try again!")
```

Unfortunately, it does not work when the player enters “3”: the type of the value returned by the `input()` function is a `string`. As a result, when we compare `playerNumber` to `magicNumber` (line 4), Python always evaluates it as `False`.

You can check it yourself in Spyder console (bottom right of the interface). Types `3 == "3"` and then press Enter:

```
In [1]: 3 == "3"
Out[1]: False
```

To compare integer values, we need to convert (or cast) the `string` into an `int`. To cast a value in Python, use the name of the type as a function. You can try it in the

Spyder console:

```
In [2]: 3 == int("3")
Out[2]: True
```

The expression `int("3")` converts the `string` “3” into an int. The expression `3 == int("3")` becomes `3 == 3`, which evaluates as `True`.

Knowing this, we can correct our first program in the following way:

```
magicNumber = 3
while True:
    word = input("What is the magic number? ")
    playerNumber = int(word)
    if playerNumber == magicNumber:
        print("This is correct! You win!")
        break
    print("This is not the magic number. Try again!")
```

If the player types “3”, the magic number is recognized, and (s)he wins.

1.5.6 What to remember

Strings: their name in Python is **str**. We can use the addition “+” or the assignment/addition “+=” operators to concatenate them. The multiplication operator "*" duplicate strings.

Booleans: their name in Python is **bool**. Their value is `True` or `False`. Conditional expressions return boolean values, using the equality operator “==”, the inequality operator “!=” or the negate operator `not`.

Integers: their name in Python is **int**. We can use the operators “+”, “", "//" *or "%" to compute sums, multiplications, divisions or remainders. For each of these operators, there is a assignment operator:* "+=", "=”, “//=” and “%=”.

The type() function: it returns the type of an expression.

Typecasting: use a Python int as a function to convert/cast a value.

```
x = int("3")   # Cast a string into an int
m = str(3)     # Cast an int into a string
```

1.6 Exceptions

1.6.1 Understanding exception messages

The program in the previous section works fine as long as the player types a string that represents an `int`. If this is not the case, the program stops, and an error message appears in the console:

```
What is the magic number? three

  File "C:\dev\magicNumber.py", line 12, in <module>
    playerNumber = int(word)

ValueError: invalid literal for int() with base 10: 'three'
```

The last line describes the problem. The first word is the name of the exception, `ValueError`. Then, we understand that an `int` casting failed.

This message appears because the expression `int(word)` raised an exception. Exceptions throwing is a usual mechanism in programming languages to manage errors.

1.6.2 Manage exceptions

We can "catch" exceptions using the `try`...`except` syntax:

```
    word = input("What is the magic number? ")
    try:
        playerNumber = int(word)
    except ValueError:
        print("Please type a number without decimals!")
        continue
```

If a `ValueError` exception is raised in the `try` block, the execution stops, and the program flow goes directly to the `except` block.

If the player types a value that can't be cast to `int`, we display the message "Please type..." and go back to the beginning of the loop (the `continue` statement).

1.6.3 Guess the number

We can get a first version of the "guess the number" game using exceptions. Remember that lines with a hash '#' are comments, and Python ignores them. We add them to ease the understanding of the program:

```
# Magic number initialization
magicNumber = 3
```

```
# Main game loop
while True:
    # Player input
    word = input("What is the magic number? ")

    # Quit if the player types "quit"
    if word == "quit":
        break

    # Int casting with exception handling
    try:
        playerNumber = int(word)
    except ValueError:
        print("Please type a number without decimals!")
        continue

    # Cases that stops the game
    if playerNumber == magicNumber:
        print("This is correct! You win!")
        break

    # Wrong number, continue
    print("This is not the magic number. Try again!")

# End of the program
```

1.6.4 What to remember

Exception messages: The last line of the exception message is the name of the exception and a description of the problem. The line just before is the location of the problem. In Spyder, click it to focus on this line in the edition window.

Exception handling: surround lines that can raise an exception with a `try` and `except`:

```
try:
    <lines that can raise an exception>
except <exception name>:
    <executed if <exception name> was raised>
```

If the `try` block can raise different exceptions, we can add more `except` blocks.

1.7 Random numbers

We use random numbers to change the magic number every time we launch the game.

1.7.1 Library import

In the previous programs, we used the `print()` and `input()` functions. They are part of the Python language, so nothing is required to use them. For other functions, like random number generation, we have to tell Python we want to use them.

For instance, to use random number functions, we need to import the `random` package at the beginning of our program:

```
import random
```

Note that if you type this in Spyder, you see a warning telling you don't use the `random` package. It is as expected, and it disappears once you use any function of this package.

The `random` package is part of the Python standard library, so it is always available when we ask for it. The default Anaconda setup already includes a lot of useful libraries. For more specific ones, like Pygame, we will see later how to proceed.

1.7.2 Generate a random integer

We can generate a random integer in the following way:

```
magicNumber = random.randint(1,10)
```

We use the `randint()` function inside the `random` package. To tell that this function is in the `random` package, we type the package's name before it, followed by a dot.

This line generates a random integer between 1 and 10 and puts it in `magicNumber`. If we replace the initialization in the game in the previous section with this one, the game now runs with a different magic number every time.

1.7.3 Import a specific function

If we don't want to type the package name before the function, we can use the following syntax:

```
from random import randint
magicNumber = randint(1,10)
```

The first line indicates that we want to use the `randint()` function from the `random` package. Then, if the `randint()` function is found in the following lines (of the same file), Python assumes it is from the `random` package. We can replace `import` random with from random import randint at the beginning of the program if randint() is the only function we need.

1.7.4 Import all symbols

Python also offers the following syntax to import all symbols from a package:

```
from random import *
```

All functions, including randint(), are available without typing the package name. **Do not use this syntax**; it is hazardous since no one knows all the package's symbols. You can have conflicts if two packages use the same function names. And trust me, if it happens in a large application, I wish you good luck to find it :)

1.7.5 Clues for the player

It is not easy for the player to find the magic number. We can give some clues to help him/her. For instance, we can say if the number entered by the player is higher or lower:

```
1  import random
2
3  magicNumber = random.randint(1,10)
4  while True:
5      word = input("What is the magic number? ")
6      if word == "quit":
7          break
8
9      try:
10         playerNumber = int(word)
11     except ValueError:
12         print("Please type a number without decimals!")
13         continue
14
15     # Cases
16     if playerNumber == magicNumber:
17         print("This is correct! You win!")
18         break
19     elif magicNumber < playerNumber:
20         print("The magic number is lower")
21     elif magicNumber > playerNumber:
22         print("The magic number is higher")
23
24     print("This is not the magic number. Try again!")
```

We added two more tests in lines 19-22. In lines 19-20, we test if the magic number is lower than the entered number. If it is the case, we print the message "The magic number is lower". Lines 21-22 handle the other case similarly. Note the `elif` statement: we must use it after an `if` statement, and we can add as much as we need. Python executes the code corresponding to the first condition evaluated as `True`, and ignores the other.

1.7.6 Count the guesses

We can count the guesses as a score. We need an integer variable to store this count. We initialize it at the beginning of the program:

```
guessCount = 0
```

Then, after every player input, we increase this counter by one:

```
guessCount = guessCount + 1
```

Python understands most arithmetic expressions, here the addition between two numbers: `guessCount + 1`.

There is a more compact syntax to increment a number:

```
guessCount += 1
```

At the end of the game, we display a message with the number of guesses:

```
print("You guessed the magic number in {} steps.".format(guessCount))
```

As we saw before, Python replaces the curly brackets `{}` with the argument of the `format()` function following the message. Don't forget the dot between the string (everything between the quotes) and the function call `format(guessCount)`.

1.7.7 What to remember

Use the keyword 'import' to access new functions: the syntax is:

```
import <package name>
```

To call a function from this package, add the name of the package and a dot:

```
<package name>.<function name>()
```

For instance:

```
import random
random.randint(1, 10)
```

Import a specific function:

```
from <package name> import <function name>
```

With this syntax, we no more need to add the name of the package, for instance:

```
from random import randint
randint(1, 10)
```

If we need to import more than one function, we can use comas:

```
from <package name> import <function name1>, <function name2>, ...
```

Import all functions: DO NOT USE IT! We can import all functions of a package with the following syntax:

```
from <package name> import *
```

I strongly advise you not to use this syntax, as it can lead to severe issues.

1.8 Refactoring with functions

It is now time to organize/refactor our program! For beginners, this step could seem strange since the refactored code does the same as before. However, refactoring is the only way to create a code easy to maintain and expand.

In software design, refactoring usually means split the program into as independent parts as possible. The first feature in the programming language we can use for that is functions.

1.8.1 Define a function

In the previous programs, we used existing functions like `print()` or `input()`. We can also create our functions using the `def` statement:

```
def printHello():
    print("Hello!")
```

The syntax is:

```
def <function name>(<arguments>):
    <function body>
```

The rules for function names are the same as for the variable name. The arguments are the function's parameters; in the `printHello()` example, there are no arguments. The function body is a usual Python code block defined using indentation.

1.8.2 Call a function

To call a function, we use its name and parenthesis, as we did before with functions like `print()`:

```
printHello()
```

If you run this program (with the function definition before!), it executes the code inside the function body and prints "Hello!".

1.8.3 Refactor the game with functions

In the previous posts, we created the "guess the number" game. We can refactor it using functions to get a better organization and improve the readability of the code.

In software design, "code refactoring" is the process of restructuring an existing code. The aim is to improve the code presentation but not its behavior. It is hard to prove in a few words why refactoring is essential in software design. It is the sum of thousands of little reasons that motivate it, each of which is the result of personal experiences. So, if you don't understand why, please trust the many programmers that bled on unstructured code and hear them telling you how much this process matters!

A first refactoring we can do is embed the game into a single function:

```
def runGame():
    magicNumber = random.randint(1,10)
    guessCount = 0
    while True:
        word = input("What is the magic number? ")
        if word == "quit":
            break

        try:
            playerNumber = int(word)
        except ValueError:
            print("Please type a number without decimals!")
            continue

        guessCount += 1
        if playerNumber == magicNumber:
            print("This is correct! You win!")
            print("You guessed the magic number in {} steps.".format(guessCoun
            break
        elif magicNumber < playerNumber:
            print("The magic number is lower")
        elif magicNumber > playerNumber:
            print("The magic number is higher")

        print("This is not the magic number. Try again!")
```

Thanks to this refactoring, we can run the game by calling the `runGame()` function. That means that if we want to create a menu to start new games, the code is easier to read. For instance:

```
import random

def runGame():
    ... the game code ...

while True:
    print("Game menu")
    print("1) Start a new game")
    print("2) Quit")
    choice = input("Enter your choice:")
    if choice == "1":
        runGame()
    elif choice == "2":
        break
```

The game menu is straightforward to understand, even for a developer that never saw it: option “1” runs the game, and option “2” leaves it.

Also, note that we choose function names that describe as much as possible the task. For example, we can’t be more explicit in our game for the `runGame()` function: we ask to *run* the *game*!

1.8.4 More refactoring with the return statement

We can continue the refactoring with the creation of a function `askPlayer()` for the management of the player input:

```
def askPlayer():
    while True:
        word = input("What is the magic number? ")
        if word == "quit":
            return None
        try:
            playerNumber = int(word)
            break
        except ValueError:
            print("Please type a number without decimals!")
            continue

    return playerNumber
```

The main purpose of this function is to get a number from the player and handle cases like bad numbers. We copy all the code related to this task and put it in a `while` loop. Then, we repeat the input until the player enters a number or “quit”.

If the player enters “quit”, the function returns None. The `return` statement stops the function: Python no more executes code lines after this statement. About `None`, it is a special Python symbol that means “nothing” or “empty”. We use it rather than “quit” because we want to change how the player can stop the game without changing the way `askPlayer()` works.

At the end of the function, there is a `return` statement. It returns the value of the `playerNumber` variable. We need to return a value because Python defines variables inside function blocks. For instance, `playerNumber` in `askPlayer()` only exists in this function. If you try to access it in another function, it does not work, even if you create a variable with the same name.

For instance, if you update the `runGame()` function to use the `askPlayer()` function in the following way:

```
def runGame():
    magicNumber = random.randint(1,10)
    guessCount = 0
    while True:
        askPlayer()
        if playerNumber is None:
            break
        ...
```

Spyder tells you that `playerNumber` is undefined. As a result, if we run the program, Python raises an exception.

The right way to use the function is to copy the value returned by `askPlayer()` into a variable of `runGame()`:

```
1 def runGame():
2     magicNumber = random.randint(1,10)
3     guessCount = 0
4     while True:
5         playerNumber = askPlayer()
6         if playerNumber is None:
7             break
8         ...
```

With this code, the value of `playerNumber` of `askPlayer()` is copied into the `play-erNumber` of `runGame()` (line 5). There is two `playerNumber` variables: one in `askPlayer()` and one in `runGame()`.

Line 6 checks if `playerNumber` is a `None`. We use the `is` operator instead of a an equal operator == because we want to check that the variable references nothing (the objects could redefine the equality, which is ignored when we use `is`). If it is not clear, don't worry: just remember that, when we need to check if a variable points nothing, we use the expression `variable is None`.

1.8.5 What to remember

Refactoring: this is an essential process in software design that reorganizes the code without changing its behavior. In cases where the program is a large block, good refactoring is splitting the code into as independent parts as possible.

Function definition: the syntax is:

```
def <function name>(<arguments>):
    <function body>
```

We call our function as we call the built-in functions, like `print()` or `input()`.

The return statement: it stops the function and optionally returns values:

```
def <function name>(<arguments>):
    ...
    return <value1>, <value2>, ...
    ...
```

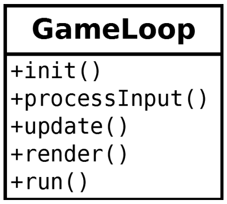
is: this operator compares the references of two variables. It is different from the equal operator “==” that compares the values of two expressions.

None: a keyword that means “nothing” or “empty”.

1.9 The Game Loop pattern

The time has come to see our first design pattern: the Game Loop Pattern! This pattern can give us many good ideas to refactor our game in a very effective way.

There is several version of the Game Loop pattern. Here we see a simple case with a single thread. The following five functions form this pattern:



The `init()` function is called at startup to initialize the game and its data. In the following, we name this data the *game state*.

The `processInput()` function is called at each game iteration to manage the controls (keyboard, mouse, pad).

The `update()` function changes the game state. The result of `processInput()` is used to update the game state and automatic processes.

The `render()` function handles display. It “converts” the game state into visual content.

The `run()` function contains the game loop. In many cases, this loop looks like this one:

```python
def run():
    init()
    while True:
        processInput()
        update()
        render()
```

As for all patterns, the Game Loop pattern is a recipe that gives you ideas on how to solve a problem. There is no unique way to use it: it depends on cases.

Following a pattern forces you to consider problems you might not be thinking about. For instance, splitting user input, data updates, and rendering is not the first thing that came to mind when we create the “guess the number” game. However, according to experimented developers who already created many games, this splitting is essential. So, right now, as beginners, we follow this advice, and later we will understand why it matters. And please trust me: the day you see it all, you will get amazed by all the genius behind these ideas!

1.9.1 The init() function

Let’s start the pattern with the `init()` function:

```python
def init():
    return "init", random.randint(1,10)
```

This function returns an initial game state. We name game data “the game state” because we see games as finite state machines. For this game, the state contains:

- The game status: this is the general status of the game, represented by a string:
  - “init”: the game starts;
  - “win”: the player wins, and the game is over;
  - “end”: the player leaves the game;
  - “lower”: the player is still playing and propose a number lower than the magic one;
  - “higher”: same but for a higher number.
- The magic number: the number the player has to find.

Bundle all game data is an important task; we’ll see more details in the following chapters.

1.9.2 The processInput() function

```python
def processInput():
    while True:
        word = input("What is the magic number? ")
        if word == "quit":
            return None

        try:
            playerNumber = int(word)
            break
        except ValueError:
            print("Please type a number without decimals!")
            continue

    return playerNumber
```

This function asks the player for a number. It handles all problems related to user input, like checking that the entered number is correct. It returns the number, or `None` if the player wants to stop the game.

For users of this function, it is like a magic box that returns instructions from the player. It does not matter how they are collected. It could be from a keyboard, a mouse, a pad, the network, or even from an AI.

1.9.3 The update() function

The `update()` function updates the game state using the player's instructions:

```python
def update(gameStatus, magicNumber, playerNumber):
    if playerNumber is None:
        gameStatus = "end"
    elif playerNumber == magicNumber:
        gameStatus = "win"
    elif magicNumber < playerNumber:
        gameStatus = "lower"
    elif magicNumber > playerNumber:
        gameStatus = "higher"

    return gameStatus, magicNumber
```

In our case, the player's instruction is `playerNumber`, and the game status and the magic number form the game state. The function updates the game status depending on the value of `playerNumber`.

Note that we don't use `gameStatus` as an input and never change the value of `mag-icNumber`. So, we could think that we can remove `gameStatus` from the arguments and `magicNumber` from the return values. Except if this is the last version of this game and we must reduce computational complexity, this is not a good idea. Maybe in future improvements of the game, we will need to update the game according to `gameStatus` or change the value of `magicNumber`. This current definition of inputs and outputs is robust for the software designer and has no reason to change.

1.9.4 The render() function

The render() function displays the current game state. It should work whatever happens, to always give a clear view of the game:

```python
def render(gameStatus, magicNumber):
    if gameStatus == "win":
        print("This is correct! You win!")
    elif gameStatus == "end":
        print("Bye!")
    elif gameStatus == "lower":
        print("The magic number is lower")
    elif gameStatus == "higher":
        print("The magic number is higher")
    else:
        raise RuntimeError("Unexpected status {}".format(gameStatus))
```

The input of this function is the game state and has no output. The process is simple: display a message according to the value of `gameStatus`.

Note that we also handle the case where `gameStatus` has an unexpected value. It is a good habit; it greatly helps the day you update the game and forget to update some parts.

1.9.5 The runGame() function

The `runGame()` function is the core of the game and uses all the previous functions:

```python
def runGame():
    gameStatus, magicNumber = init()
    while gameStatus != "win" and gameStatus != "end":
        playerNumber = processInput()
        gameStatus, magicNumber = update(
            gameStatus, magicNumber, playerNumber
        )
        render(gameStatus, magicNumber)
```

You can see the flow:

- The `init()` function returns an initial game state;
- The `processInput()` function collects instructions from the player;
- The `update()` function uses the instructions to update the game state;
- The `render()` function displays the game state.

1.9.6 What to remember

Design patterns: they are recipes created by programmers to help us find the best way to organize our code.

The Game Loop pattern: the core of all games. As for all patterns, there are several versions; the one of this chapter is a simple version based on functions. The main idea is to split input handling, game updating, and rendering.

The game state: it contains all the data of a game. It does not have data for other components. For instance, we don't store in the game state what is related to the user interface, like keys or mouse clicks.

CHAPTER 2

2D graphics with Pygame

In the first chapter, we saw the basics of Python and a first pattern. We continue discovering new Python features and patterns in this chapter while using Pygame to create games.

You can find examples from this chapter here: `https://github.com/philippehenri-gosselin/discoverpythonpatterns/tree/master/chap02`. The names of the examples begin with the corresponding section. All the tileset images are in this folder.

2.1 First steps

2.1.1 Install Pygame

Pygame is a popular library for creating 2D games. Like the random library we used before, the `import` statement allows us to access all its features. Unfortunately, the Anaconda environment we installed in the previous chapter does not embed Pygame, and we have to install it.

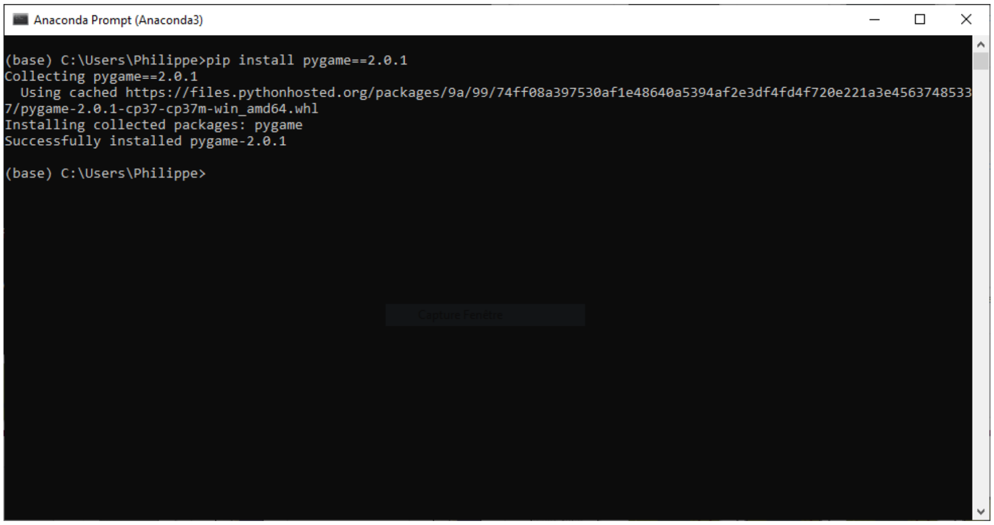
To install a new python package, we need an Anaconda prompt. In Windows, type “Anaconda” in the start menu and click “Anaconda Prompt”. Then type the following command:

```
pip install pygame==2.6.1
```

We add `==2.6.1` after `pygame` to ask for Pygame version 2.6.1. It ensures that all the following code examples will work. You should always install packages with identical versions; only upgrade if needed. Function names and arguments used to change often in non-standard packages.

If you are not using Python version 3.11, you may face problems with the installation of this version of Pygame. If it is the case, you can either switch to Python 3.11 or try a more recent version of Pygame.

Running this command, you should get something similar to:



```
(base) C:\Users\Philippe>pip install pygame==2.0.1
Collecting pygame==2.0.1
  Using cached https://files.pythonhosted.org/packages/9a/99/74ff08a397530af1e48640a5394af2e3df4fd4f720e221a3e4563748533
7/pygame-2.0.1-cp37-cp37m-win_amd64.whl
Installing collected packages: pygame
Successfully installed pygame-2.0.1

(base) C:\Users\Philippe>
```

2.1.2 First Pygame program

Here is a basic Pygame program that displays a window and exits if we close it:

```
1 import pygame
2 pygame.init()
3 window = pygame.display.set_mode((640,480))
4 while True:
5     event = pygame.event.poll()
6     if event.type == pygame.QUIT:
7         break
8     pygame.display.update()
9 pygame.quit()
```

The first line imports the Pygame library. It gives access to all functions in the library. We can call all these functions preceding their name with `pygame` and a dot. For example, line 2 calls the `init()` function that initializes the Pygame library. As for the `random` library, we could also use the following syntax:

```
from pygame import init
init()
```

With the `from ... import` syntax, we don't need to type `pygame.` before the function's name. However, we must be sure that `init()` is the only function with that name: it is a common function name. Therefore, we should use the extended syntax with these functions.

Line 3 creates a window with a size of 640 per 480 pixels. It is the size of the area where we draw our game, not the decorations

Lines 4-9 are the main loop. This loop should look familiar since we saw the Game Loop pattern in the previous chapter. Lines 5-7 are the processing of inputs, and line 8 is the rendering. Pygame stores all events (mouse moves, keyboard presses, etc.) in a queue. Then, we can read this queue using the function `pygame.event.poll()`. Note that each time we get an event, Pygame removes it from the queue (which is the expected behavior of a queue, but it is another story!).

The only event we process in this example is the one that asks for the end of the game. We read the `type` attribute of the event returned by `pygame.event.poll()`. These types are always values like `pygame.XXX` (or only XXX if we did a `from pygame import XXX`). We will see many examples throughout the book.

2.1.3 The for statement

We process only one event per update with the previous program, leading to unwanted latencies. We can ask for all the currently available events using the function `pygame.event.get()`:

```
running = True
while running:
    eventList = pygame.event.get()
    for event in eventList:
        if event.type == pygame.QUIT:
            running = False
    pygame.display.update()
pygame.quit()
```

In the program above, we store the list of events into the variable `eventList`. Note that `pygame.event.get()` removes all current events, so we'll get new ones next time.

To read all elements of a list, we can use the `for` statement. The syntax is straight-forward:

```
for item in list:
    ...block...
```

`item` is the variable's name that contains one element in `list` (a copy or a reference). For instance, if we run the following program:

```
list = [1,2,3]
for item in list:
    print(item)
```

It displays:

```
1
2
3
```

Python repeats the `for` block for each value in the list, and `item` gets one of its values. Furthermore, the order is always the same: from the first to the last element.

In the case of the Pygame event list, for each event in the list, if the event type is `pygame.QUIT`, we set the variable `running` to `False` to stop the main loop:

```
    for event in eventList:
        if event.type == pygame.QUIT:
            running = False
```

Note that we can't use the `break` statement to stop the main loop in the `for` loop. So, using the `break` statement in the `for` loop ends the `for` loop, but not the main loop that includes it. As a result, we use a variable as we did before in the "guess the number" game.

2.1.4 What to remember

Install a package: open an Anaconda prompt, and type:

```
pip install <package>==<version>
```

`<package>` is the name of the package (like “pygame”), and `<version>` is the version we want (ex: 2.0.1). Specifying the version is not mandatory but highly recommended.

Pygame main loop: it follows the Game Loop pattern:

```
running = True
while running:
    # Input handling
    eventList = pygame.event.get()
    for event in eventList:
        ...process event...
    # Game updating
        ...update game state...
    # Rendering
    pygame.display.update()
```

The for statement: the basic syntax is:

```
for <variable> in <container>:
    ...block...
```

`<variable>` is the variable’s name that contains one item in the `<container>`. The container can be any variable or expression that holds data like a list, a dictionary. We will see more about that in the following sections.

2.2 Basic drawing

Pygame offers many functions for drawing; we see a couple of examples here.

2.2.1 Draw a pixel

We can draw a pixel using the `set_at()` method of a surface:

```
<surface>.set_at(<position>, <color>)
```

Methods are like functions, except we need an object to use them. In this case, `<surface>` is a Pygame surface on which we can call the `set_at()` method.

Let's see an example:

```
window.set_at((320, 240), (255, 255, 255))
```

The `window` variable is the surface we create at the beginning of our program with the `pygame.display.set_mode()` function.

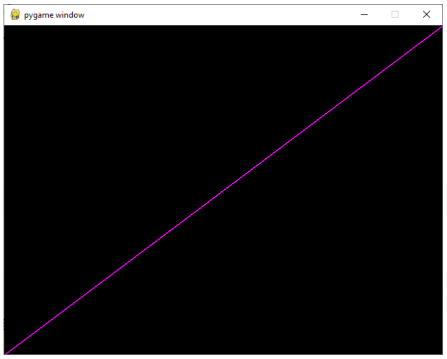
The expression `(320, 240)` is a tuple. It is a standard Python data type that combines several values. In this case, it is a tuple of two integers with the coordinates of a pixel. Since our window area has 640 per 480 pixels, it is in the center.

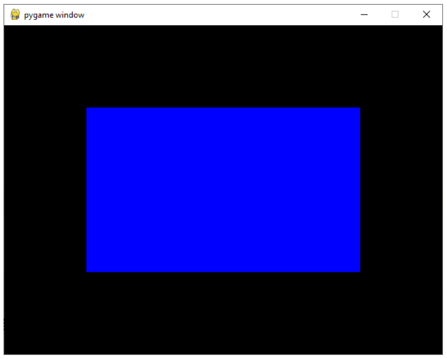
The expression `(255, 255, 255)` is a tuple of three integers. It contains the red, green, and blue values of a color. Values are between 0 (no color) and 255 (maximum). For instance `(255, 0, 0)` is the red color, and `(255, 0, 255)` is the purple color. You can test many color combinations in Paint for Windows and most image editors.

We can get the same result using variables:

```
position = (320, 240)
color = (255, 255, 255)
window.set_at(position, color)
```

If we add the pixel drawing right before the `pygame.display.update()` call, we get a white pixel in the center of the window.





Draw a line (left) and a rectangle (right).

2.2.2 Draw a line

We draw a line with the `pygame.draw.line()` method:

```
pygame.draw.line(<surface>, <color>, <start>, <end>, <width>)
```

`<surface>` is the surface on which we draw; `<color>` is the (red,green,blue) color; `<start>` are the (x,y) coordinates of the first edge of the line; `<end>` are the (x,y) coordinates of the second edge of the line; `<width>` is the line thickness.

With the following values, we get a diagonal purple line:

```
pygame.draw.line(window, (255, 0, 255), (0, 480), (640, 0), 2)
```

2.2.3 Draw a rectangle

We draw a rectangle with the `pygame.draw.rect()` method:

```
pygame.draw.rect(<surface>, <color>, <rect>)
```

`<rect>` is a tuple with four integer values: the top left x,y coordinates, the width, and the height of the rectangle. For instance `(10,20,200,100)` is a rectangle with x=10, y=20, width=200 and height=100.

We can use this function in our program to draw a blue rectangle in the center of the window:

```
pygame.draw.rect(window, (0, 0, 255), (120, 120, 400, 240))
```

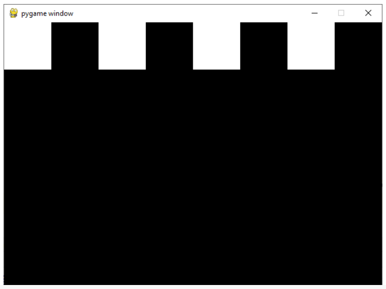
2.2.4 Draw a checkerboard

We can not draw everything with a single function with Pygame; sometimes, we have to combine calls. Here is an example with the drawing of a checkboard.

2.2.4.1 Repeat code the right way

The first approach is to repeat the drawing manually. For instance, we can render the first line of the checkerboard in the following way:

```
pygame.draw.rect(window, (255, 255, 255), (0, 0, 80, 80))
pygame.draw.rect(window, (255, 255, 255), (160, 0, 80, 80))
pygame.draw.rect(window, (255, 255, 255), (320, 0, 80, 80))
pygame.draw.rect(window, (255, 255, 255), (480, 0, 80, 80))
```



It works fine: we see the first row of the checkerboard. However, we have to compute all coordinates manually. It also leads to a lot of code. But the biggest problem is that it can only render a specific size. In this example, we draw four rectangles that fit our window, but we must recompute all coordinates if we want more rectangles.

A better approach is to use the `for` statement:

```
for x in range(4):
    pygame.draw.rect(window, (255, 255, 255), (x * 160, 0, 80, 80))
```

Note the `range()` function: it tells the `for` statement that we want to iterate integers from 0 to 3. The first value is zero, and the last is the value in `range()` minus one. So in this example, x gets the following values: 0, 1, 2, and 3.

The code is easier to understand: experimented developers immediately know that we draw four rectangles shift by 160 pixels.

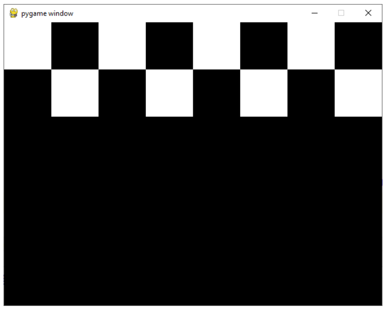
Furthermore, it is easier to change the number of rectangles. For instance, if we switch to eight rectangles:

```
for x in range(8):
    pygame.draw.rect(window, (255, 255, 255), (x * 80, 0, 40, 40))
```

2.2.4.2 Code readability

To add a second row, we can duplicate our `for` loop and shift the coordinates:

```
for x in range(4):
    pygame.draw.rect(window, (255, 255, 255), (x * 160, 0, 80, 80))
for x in range(4):
    pygame.draw.rect(window, (255, 255, 255), (x * 160 + 80, 80, 80, 80))
```



An essential point in software development is code readability. Unfortunately, many developers neglect it: once the code works, they don't touch it anymore as it could break anytime. It is to save a little time and lose a lot afterward. Except if we have to send our code to our client immediately or if the code is only a proof of concept, there is no reason not to spend a moment improving it.

What makes a code more readable is its ability to be understood by any developer. In this example, developers likely do not know the arguments of the `pygame.draw.rect()` function. We can help them with variables:

```
color = (255, 255, 255)
width = 80
height = 80
for x in range(4):
    rect = (x * 2 * width, 0, width, height)
    pygame.draw.rect(window, color, rect)
for x in range(4):
    rect = (x * 2 * width + width, height, width, height)
    pygame.draw.rect(window, color, rect)
```

Now it is evident that the arguments are: a surface, a color, and a rectangle. Furthermore, the rectangle has an x coordinate, a y coordinate, a width, and a height.

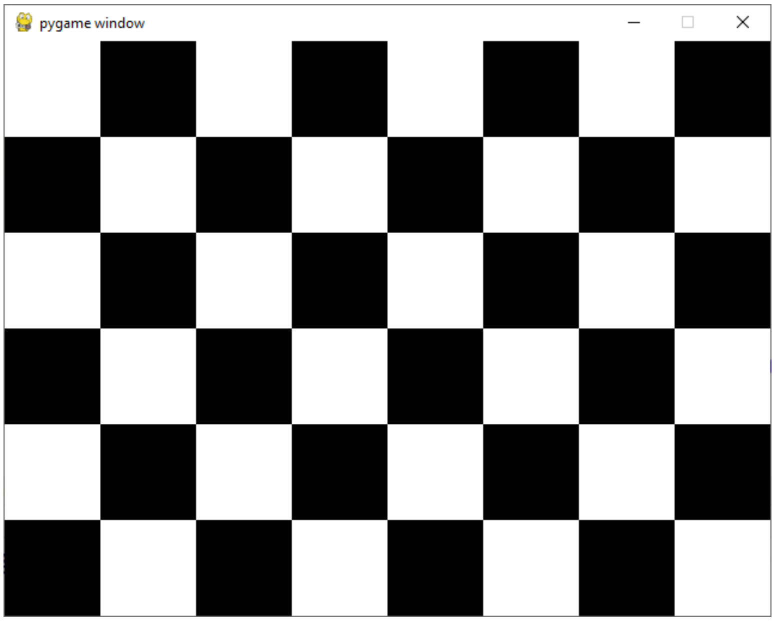
The other advantage of this minor refactoring is that it is easier to change the color and the size of the rectangles. We could also compute the number of rectangles we can draw: it is a good exercise! The solution is in the code that comes with this book; try different window sizes to see what happens.

2.2.4.3 Final checkerboard

To get the entire checkerboard, we need to repeat three times the two rows:

```python
color = (255, 255, 255)
width = 80
height = 80
for y in range(3):
    for x in range(4):
        rect = (x * 2 * width, y * 2 * height, width, height)
        pygame.draw.rect(window, color, rect)
    for x in range(4):
        rect = (x * 2 * width + width, y * 2 * height + height,
                width, height)
        pygame.draw.rect(window, color, rect)
```

We get our complete cherkerboard:



2.2.5 What to remember

Pygame draw functions:

- `<surface>.set_at(<position>, <color>)` draws a pixel;
- `pygame.draw.line(<surface>, <color>, <start>, <end>, <width>)` draws a line;
- `pygame.draw.rect(<surface>, <color>, <rect>)` draws a rectangle.

The arguments are:

- `<position>`, `<start>` and `<end>` are tuples of two integers like `(<x>, <y>)`;
- `<color>` is a tuple of three integers like `(<red>, <green>, <blue>)`;
- `<width>` is an integer;
- `<rect>` is tuple of four integers like `(<x>, <y>, <width>, <height>)`.

Pygame documentation describes similar functions here:

`https://www.pygame.org/docs/ref/draw.html`

Tuples: they contain several values. Use parentheses and coma to create tuples:

```
single = (23)
two = (12, 18)
three = (8, 3, 9)
...
```

On the contrary to a list, values in a tuple can not change. We will see more about tuples in the following sections.

Iterate over integers: we can iterate between 0 and a maximum value minus one using `for` and `range()`:

```
for <variable> in range(<maximum>):
    ...
```

`<variable>` gets values `0, 1, 2, ..., <maximum> - 1`.

Code readability: the golden rule is: the more someone can understand a code (s)he has never seen, the more readable it is.

When a code is running fine, always take a moment to see if you can improve its readability. Even if you are the only one creating the game, it will help your future “you” understand the code you wrote long ago!

2.3 Window and keyboard

This section introduces controls with the keyboard and some improvements like window centering and frame rate handling.

2.3.1 Keyboard events

Pygame events allow the capture of keyboard presses:

```python
import pygame

pygame.init()
window = pygame.display.set_mode((640, 480))

x = 120
y = 120
running = True
while running:

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
            break
        elif event.type == pygame.KEYDOWN:
            if event.key == pygame.K_ESCAPE:
                running = False
                break
            elif event.key == pygame.K_RIGHT:
                x += 8
            elif event.key == pygame.K_LEFT:
                x -= 8
            elif event.key == pygame.K_DOWN:
                y += 8
            elif event.key == pygame.K_UP:
                y -= 8

    window.fill((0, 0, 0))
    pygame.draw.rect(window, (0, 0, 255), (x, y, 400, 240))
    pygame.display.update()

pygame.quit()
```

This program draws a blue rectangle at coordinates (x,y) with a width of 400 pixels and a height of 240 pixels. We can move it using the arrows and quit the game using the Espace key or closing the window.

Lines 15-26 handle the keyboard. Python executes these lines if `event.type` is not `pygame.QUIT` (the condition at line 12 is `False`) and if `event.type` is `pygame.KEYDOWN` (the condition at line 15 is `True`). “Key down” is the event where a key goes from unpressed to pressed. It is different from holding or releasing a key.

Once we know that it is a `pygame.KEYDOWN` event (line 15), the `key` attribute of `event` contains the value of the pressed key. These values are always something like `pygame.K_XXX`. In Spyder, if you type `pygame.K_`, and then hit Ctrl+Space, you will see the list of all key names.

The Espace key quit the game (like the `pygame.QUIT` event). The arrow keys change the x and y coordinates of the rectangle.

Line 28 calls the function `window.fill()` with a black color `(0,0,0)`. It fills all the window client area with a color. You can try with a different color, for instance, red `(255,0,0)`, green `(0,255,0)` or blue `(0,0,255)`.

2.3.2 Window centering

When the previous program starts, the window can appear at different locations on the screen. We can force the centering by adding the following commands at the beginning of the program:

```
import os
os.environ['SDL_VIDEO_CENTERED'] = '1'
```

The first line imports the standard package `os`. It contains many functions related to the operating system.

The second line sets the environment variable `SDL_VIDEO_CENTERED` to “1” (NB: the character “1” not the number). Pygame is based on the SDL library, and this one uses environment variables to define some options like window centering. You can do more hacks like this one; see the SDL document at https://www.libsdl.org.

2.3.3 Clock and frame rate

If you look at the CPU usage when the program runs, you can see that it uses 100% of a core. As a result, the program renders much more frames than what a screen can display.

To save computations, we can use a `pygame.tick.Clock` to limit the frame rate. First of all, we need to create such an object at the beginning of the program:

```
clock = pygame.time.Clock()
```

Note that `pygame.tick.Clock()` looks like a function call, and the line above puts the return value of this function into the variable `clock`. Actually, it is not exactly a function call: it creates an instance of the `pygame.time.Clock` class. Class instances are advanced objects; for example, they can have several sub-variables called attributes. We see them in the next section.

Then, at the end of the `while` loop, add the following line:

```
clock.tick(60)
```

It limits the frame rate to 60 per second. More specifically, it releases the CPU such that the frame rate is 60 per second. So, for instance, if the beginning of the `while` took five milliseconds, then it releases the CPU for 11 milliseconds (at 60 frames per second, we have 16 milliseconds per frame).

Considering the function call, it should look strange to you: `clock` is a variable, not a package like `random` or `pygame`. It works because `clock` refers to a class instance. These objects can contain functions called methods. So, `tick()` is a method of the `pygame.time.Clock` class. It has two arguments: `clock` and 60. The first argument is the class instance, and all the others are in the parentheses. Don't worry if this is not clear; we will detail that later.

2.3.4 Window caption and icon

To finish improvements, we set a window caption (or title) and an icon. For the first one, if we want to see "My game" in the window title, we have to call `pygame.display.set_caption()` after the creation of the window:

```
pygame.display.set_caption("My game")
```

For the window icon, we first need an image. We can load one using the `pygame.image.load()` function:

```
iconImage = pygame.image.load("icon.png")
```

It assumes that the file "icon.png" is in the folder where our program is running.

Then, the window icon is set using the `pygame.display.set_icon()` function:

```
pygame.display.set_icon(iconImage)
```

Note that we can pack both lines into a single one (the use of a variable is not mandatory):

```
pygame.display.set_icon(pygame.image.load("icon.png"))
```

2.3.5 What to remember

Pygame key events: if the type of an event is `pygame.KEYDOWN` (or `pygame.KEYUP`), then the player pressed (or released) a key. In these cases, the `key` attribute of the event contains the key code, which is something like `pygame.K_xxx`.

Erase a surface: if we want to fill a surface with the same color, we can use the `fill()` method:

```
<surface>.fill(<color>)
```

It is a method: we need a surface to call it.

Center window at startup: import the `os` package and set the environment variable `SDL_VIDEO_CENTERED` to “1”:

```
os.environ['SDL_VIDEO_CENTERED'] = '1'
```

`os.environ` is a dictionary; we will see them later.

Set the window caption: call this after the creation of the window:

```
pygame.display.set_caption(<string>)
```

Set the window icon: also call after the window creation:

```
pygame.display.set_icon(pygame.image.load(<image file>))
```

The image file must be in the directory of our Python script.

2.4 The Game Loop pattern with a class

It is time to see one of the main components of software design: object programming. There are several ways to implement it; in this section and in the following, we use classes.

We use these classes to implement the Game Loop pattern and refactor the code to get a more robust and readable program.

2.4.1 Create a class with Python

In Python, we can create a class using the following syntax:

```
class MyClass:

    def __init__(self, value):
        self.value = value

    def incValue(self):
        self.value += 1
```

The name of this class is `MyClass`. We can choose any available name as usual. Note that a lot of programmers start class names with capital letters.

This class has two methods: `__init__()` and `incValue()`. Methods are like functions, except that they are bind to an object.

The first argument is always a reference to a class instance, and most Python programmers name it `self`. A class instance is an object of a class: it contains all the attributes and methods of the class.

The body of methods is like that of functions: the following code block defined by the indentation contains all the lines of the method.

2.4.2 Constructor and attributes

The `__init__()` method is a unique method called the constructor. Python calls it when we create a new class instance. As we saw before, we can create a class instance by using the class name as a function:

```
myClass = MyClass(10)
```

Note that there is only one argument to this call: it is as if we are ignoring the `self` argument of the `__init__()` method (or constructor).

Inside the `__init__()` method, we create an attribute `value` using the dot operator:

```
    def __init__(self, value):
        self.value = value
```

The expression `self.value = value` creates the attribute with an initial value `value`. Since Python is a dynamic language, we can create an attribute in any class method and outside the class. However, I do not recommend it! Please code as if it was impossible until you get a strong knowledge of Python. Only create attributes in the constructor.

Once an attribute exists, we can access it using the dot operator:

```
myClass = MyClass(10)
print(myClass.value)
```

This program displays “10” in the console.

2.4.3 Method call

To call a class method, we need a reference to a class instance, and then use the dot operator:

```
myClass = MyClass(10)
myClass.incValue()
print(myClass.value)
```

This program displays “11” in the console.

Note that, as for the constructor, we ignore the first argument `self`. The class instance before the dot defines this value. There is a function call equivalent of a method call, for the method `incValue()` it is:

```
MyClass.incValue(myClass)
```

We can see the use of the first argument that sets `self` to `myClass`. Thus, a method call is a syntax improvement that eases coding and reading.

2.4.4 Game Loop pattern

Using a class, we can get a better implementation of the Game Loop pattern:

```
class Game():
    def __init__(self):
        ... Initialization ...

    def processInput(self):
        ... Handle user input ...

    def update(self):
        ... Update game state ...

    def render(self):
        ... Render game state ...

    def run(self):
        ... Main loop ...
```

The class contains all the methods related to the pattern. Furthermore, we no more need to add many arguments to these methods since all data is in the class attributes.

Let's now see the implementation of all these methods, which use the previous section's code with a different organization.

2.4.4.1 The init() method

The `__init__()` method contains all the code at the beginning of the previous program, except that data is put in the class attributes:

```
def __init__(self):
    pygame.init()
    self.window = pygame.display.set_mode((640,480))
    pygame.display.set_caption("Discover Python & Patterns")
    pygame.display.set_icon(pygame.image.load("icon.png"))
    self.clock = pygame.time.Clock()
    self.x = 120
    self.y = 120
    self.running = True
```

For instance, the window handle is in the attribute `window` attribute using the expression `self.window = pygame.display.set_mode((640,480))`.

In all other methods, if we type `self.window`, we get the value of the `window` attribute.

2.4.4.2 The processInput() method

The `processInput()` method contains the `for` loop that processes of Pygame events:

```
def processInput(self):
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            self.running = False
            break
        elif event.type == pygame.KEYDOWN:
            if event.key == pygame.K_ESCAPE:
                self.running = False
```

```
                    break
                elif event.key == pygame.K_RIGHT:
                    self.x += 8
                elif event.key == pygame.K_LEFT:
                    self.x -= 8
                elif event.key == pygame.K_DOWN:
                    self.y += 8
                elif event.key == pygame.K_UP:
                    self.y -= 8
```

It is as before, except that we set or update class attributes. For instance, `x += 8` becomes `self.x += 8`.

2.4.4.3 The update() method

In the Game Lopp pattern, the `update()` method should change the game state. In our case, this data is made of the `x` and `y` attributes. However, we already update them in the `processInput()`. As a result, there is nothing to do during the update. In Python, to tell that there is nothing to do, you can use the `pass` keyword:

```
def update(self):
    pass
```

This implementation of the Game Loop pattern is not exemplary. We do not change the previous code, except for the use of the class attributes. It would have been too complex to do everything at the same time. We will see in the next section how to get a better implementation of the Game Loop pattern.

2.4.4.4 The render() method

The `render()` method uses the data from the game state (`x` and `y` attributes) to display the current state of the game:

```
def render(self):
    self.window.fill((0,0,0))
    pygame.draw.rect(self.window,(0,0,255),(self.x,self.y,400,240))
    pygame.display.update()
```

As in the previous methods, we replace the global variables with attributes, like `window` that becomes `self.window`.

2.4.4.5 The run() method

The `run()` method contains the main loop, and calls the other methods:

```python
def run(self):
    while self.running:
        self.processInput()
        self.update()
        self.render()
        self.clock.tick(60)
```

We call a method with the dot operator. For instance, `self.update()` calls the method `update()` using the class instance referenced by `self`.

Each game step (input, update, and render) is run 60 times per second (line `self.clock.tick(60)`). It is the simplest case, but we could change that in this method. For instance, we could only render half of the frames on slow computers. That way, the game still runs the same on every computer (slow and fast ones), and we have nothing to change in the input processing and game state update. It also means that if we want to run the game on a server with no display, we only need to remove the call to the `render()` method.

2.4.4.6 Create and run the game

Finally, to run and close the game, we only need the following lines:

```python
1 game = Game()
2 game.run()
3 pygame.quit()
```

Line 1 creates a new instance of the `Game` class. It is the `Game` class used as if it was a function. The `__init__()` method has only one argument (`self`), so this call has no arguments.

Line 2 calls the `run()` method of the `game` instance.

Line 3 calls the `quit()` function of the `pygame` package to close all Pygame content.

2.4.5 What to remember

Class: we create a class with the **class** keyword:

```
class MyClass:
    ...
```

The following block contains all elements of the class.

Method: we define class methods with the **def** keyword inside a class:

```
class MyClass:
    def myMethod(self, arg1, arg2):
        ...
```

They look like functions, except for the first argument (usually named `self`) that always points to the class instance. This first argument is mandatory, and you can add as many arguments as you like.

Constructor: it is a particular method named `__init__` which Python calls when we create a class instance:

```
class MyClass:
    def __init__(self, arg1, arg2):
        ...
```

Class instance: it is an object that belongs to the class. It has all its properties (attributes, methods, . . .). We create a class instance calling the class as if it was a function:

```
instance = MyClass(value1, value2)
```

The number of arguments depends on the ones we defined in the constructor (excluding `self`).

Method call: We call a method using the dot operator:

```
instance.myMethod(value1, value2)
```

Note that there is no value for the `self` argument: it is always the instance.

Method calls are function calls; we could also use the following syntax:

```
MyClass.myMethod(instance, value1, value2)
```

2.5 A better Game Loop pattern implementation

The implementation of the Game Loop pattern in the previous post is not good: we update the game state in the `processInput()` method. The experimented programmers who created this pattern claim that we should split input handling and game updating. As before, we trust their experience, and we use the Command pattern to get this feature.

2.5.1 The Command pattern

The main idea behind this pattern is to separate the origin of the modifications from their implementation. For instance, when the player presses an arrow key, the controlled character is not moved. We first memorize that the player wants to move the character in a direction. Then, in a second step, the character is moved according to what we memorized.

It is quite a natural pattern. For instance, when you command a meal in a restaurant, the waiter notes your order on a piece of paper. Then, he/she goes to the kitchen and gives it to the cooker. The cooker is free to prepare the meals in the best order. For instance, if several customers are asking for the same meal, he/she can regroup the preparations.

In software design, this is very similar. The customer is the user (or the player in games) asking for the execution of a task. These orders, or commands, are stored in variables. Then, the code lines that execute tasks read the commands and act accordingly.

As for all patterns, there are many possible implementations. For our first implementation, we are using class attributes to store the commands. Later in the book, we will see more advanced representations.

2.5.2 Separate input handling and data updating

We first create two new attributes in the constructor to store the move direction the player is asking for:

```python
def __init__(self):
    ... the first lines are as before ...

    self.moveCommandX = 0
    self.moveCommandY = 0
```

Then, in the `processInput()` method, we initialize these two attributes with a zero value: it means that, by default, there is no movement. In the event processing loop, and more specifically in the processing of arrow keys, we no more update the `x` and `y` attributes (the rectangle's location = the game state). Instead, we now store the shift we want to apply to this location:

```python
def processInput(self):
    self.moveCommandX = 0
    self.moveCommandY = 0
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            self.running = False
            break
        elif event.type == pygame.KEYDOWN:
            if event.key == pygame.K_ESCAPE:
                self.running = False
                break
            elif event.key == pygame.K_RIGHT:
                self.moveCommandX = 8
            elif event.key == pygame.K_LEFT:
                self.moveCommandX = -8
            elif event.key == pygame.K_DOWN:
                self.moveCommandY = 8
            elif event.key == pygame.K_UP:
                self.moveCommandY = -8
```

The `update()` method has a content: it updates the location of the rectangle (`x` and `y` attributes) according to the values of the move command attributes:

```python
def update(self):
    self.x += self.moveCommandX
    self.y += self.moveCommandY
```

2.5.3 Why is it better?

If you run the program with these changes, it works as before. So you could ask yourself: why should I do this? It was working well before!

We are following a golden rule in software design: divide and conquer. It is impossible to solve complex problems directly, even for the most clever developer. There is always a level of complexity that becomes intractable. The solution is then to split a complex problem into more straightforward problems. If these sub-problems are still too complicated, we split again. And we repeat, until getting problems simple enough to be solved.

It is what the Game Loop pattern proposes to do for the very complex problem of game development. It splits it into three main sub-problems: input handling, game update, and rendering. For this to work, the splitting must be effective.

In the previous implementation, we did not separate input handling and game updating. Sooner or later, we would have faced too complex problems. Thanks to the improvement using the Command pattern, we are better protected from these future issues.

If you are still not convinced, let's consider the case of controls. In our game example, we use the keyboard arrows. But what about using a gamepad, for instance? With the first implementation, we would have to duplicate the update of the x and y attributes for each control. Our example is elementary, but it is easy to imagine more complex cases dealing with collisions. With the new implementation, we only have to create commands for each control (an easy task) and only need a single implementation of the game updating (a more complex task).

2.5.4 The game state

Now that we have introduced the "divide and conquer" golden rule, we use it as much as possible. For instance, in our current game implementation, the game state and the user interface (=input handling and rendering) are mixed.

To split these two features, we create a new class `GameState` that handles the game state (storage and update):

```python
class GameState:
    def __init__(self):
        self.x = 120
        self.y = 120

    def update(self, moveCommandX, moveCommandY):
        self.x += moveCommandX
        self.y += moveCommandY
```

In the `__init__()` constructor, we create the `x` and `y` attributes.

In the `update()` method, we update the coordinates using the move command values.

In the constructor of my main class, we replace the creation of the `x` and `y` attributes with the creation of a new `GameState` instance:

```python
def __init__(self):
    ...
    self.gameState = GameState()
    ...
```

For the `update()` method, we call the `update()` method of the `GameState` class:

```python
def update(self):
    self.gameState.update(self.moveCommandX, self.moveCommandY)
```

And finally, in the `render()` method, we use the attributes in the game state instead of the ones in the main class. For instance, `self.x` becomes `self.gameState.x`.

In this way, we delegate all storing and updating game data tasks to the “GameState” class. So the main class no more has to worry about that, and it only has to know the name of the attributes and methods it can use.

2.5.5 What to remember

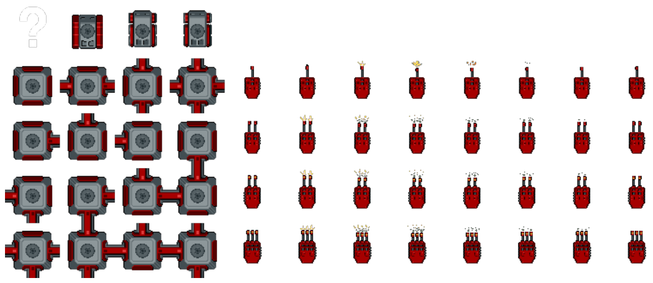
Divide and conquer: always split complex problems into sub-problems. The Game Loop pattern follows this golden rule and separates the game processing into three main steps.

The Command pattern: it collects user inputs into someplace (class attributes, instances, . . .) like a waiter in a restaurant. Several inputs can lead to the same data. For instance, the keyboard or gamepad right arrow creates the same command “move to the right”. Then, it uses the commands to update game data. For instance, if we see the “move to the right” command, we move the character to the right without knowing if it was caused by a keyboard, a gamepad, a network packet, a recording, etc.

2.6 Sprites

2.6.1 Tileset

We wish to create a “tank battle” game, so we looked for free game assets containing tank tilesets. We can find many on https://itch.io, and select this one created by zintoki: https://zintoki.itch.io/ground-shaker. We gather all the sprites we need into a single image:

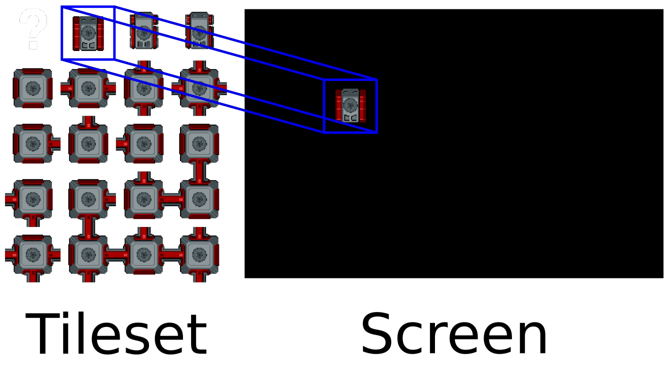


To load an image with Pygame, we can do as before with the `pygame.image.load()` function:

```
unitsTileset = pygame.image.load("units.png")
```

2.6.2 Draw a sprite from a tileset

To draw a sprite from a tileset, we copy the area with the sprite to the screen:



Any instance of the `pygame.Surface` class can do this copy using the `blit()` method:

```
surface.blit(unitsTileset, location, rectangle)
```

This method has three arguments: the tileset image, the location in the surface, and the rectangle in the tileset.

We can represent locations using tuples `(x,y)`:

```
location = (x, y)
```

Pygame represents rectangles with instances of the `pygame.Rect` class:

```
rectangle = pygame.Rect(x, y, width, height)
```

`x` and `y` are the coordinates of the top left corner and `width` and `height` the rectangle's width and height.

2.6.3 Draw a tank sprite

For the case of our tank tileset, all sprites have 64 per 64 pixels. Furthermore, to select the tank in the first line and second column, the top-left coordinates are: `x = 0` and `y = 64`. The rectangle is then:

```
rectangle = pygame.Rect(64, 0, 64, 64)
```

If we want to display the tank near the center of a window of 256 per 256 pixels, we choose coordinates `x = 96` and `y = 96`:

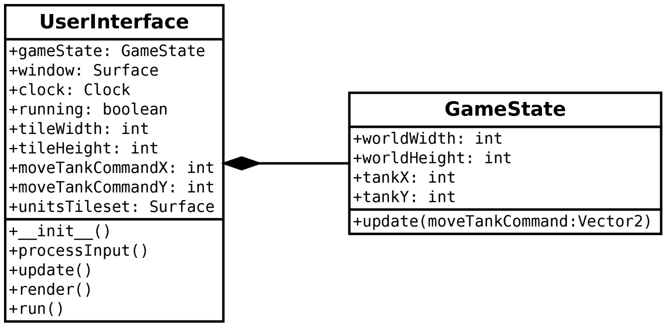
```
location = (96, 96)
```

The following program renders a tank sprite in the middle of the window:

```
import pygame

unitsTileset = pygame.image.load("units.png")
window = pygame.display.set_mode((256,256))
location = pygame.math.Vector2(96, 96)
rectangle = pygame.Rect(64, 0, 64, 64)
window.blit(unitsTileset, location, rectangle)

while True:
    event = pygame.event.poll()
    if event.type == pygame.QUIT:
        break
    pygame.display.update()

pygame.quit()
```

2.6.4 Control the tank

We update the previous program and replace the rectangle with a tank sprite. Here is the structure of this program:



Class `UserInterface`:

- New attributes `tileWidth`, `tileHeight`: defines the size of sprites. It makes a more readable code and eases the update to sprites of a different size.
- New attributes `moveTankCommandX`, `moveTankCommandY`: it replaces the previous `moveCommandX` and `moveCommandY` attributes and contains the move command for the tank.
- New attribute `unitsTileset`: contains the tileset image

Class `GameState`:

- New attributes `worldWidth`, `worldHeight`: defines the size of the world.
- New attributes `tankX`, `tankY`: defines the location of the tank and replaces the previous `x` and `y` attributes.

2.6.4.1 `UserInterface` constructor

We add new lines to create the new attributes and update the window size according to the world size:

```
def __init__(self):
    ...
    self.tileWidth = 64
    self.tileHeight = 64
    self.unitsTilset = pygame.image.load("units.png")

    windowWidth = self.gameState.worldWidth * self.tileWidth
    windowHeight = self.gameState.worldHeight * self.tileHeight
    self.window = pygame.display.set_mode((windowWidth, windowHeight))
    ...
```

The window pixel size is the world size multiplied by the tile pixel size. If the world size is (16,10) and the tile pixel size (64,64), then the world pixel size is `(16 * 64,10 * 64) = (1024,640)`.

2.6.4.2 `UserInterface` method `render()`

This method draws a tank (only the base, no gun yet):

```
1  def render(self):
2      self.window.fill((0, 0, 0))
3      spriteX = self.gameState.tankX * self.tileWidth
4      spriteY = self.gameState.tankY * self.tileHeight
5      tileX = 1 * self.tileWidth
6      tileY = 0 * self.tileHeight
7      tileRect = Rect(tileX, tileY, self.tileWidth, self.tileHeight)
8      self.window.blit(self.unitsTileset, (spriteX, spriteY), tileRect)
9      pygame.display.update()
```

The expressions at lines 3-4 are similar to the one we used to compute the window size: it multiplies the tank location by the size of a tile. Lines 5-6 compute the top-left corner of the rectangle that contains the tank sprite. Line 8 creates a rectangle that contains the tank sprite. Finally, line 8 draws the tank in the window surface.

2.6.4.3 `GameState` method `update()`

The `update()` method works as before, except that the player can not move the tank outside the world:

```
1  def update(self, moveTankCommandX, moveTankCommandY):
2      self.tankX += moveTankCommandX
3      self.tankY += moveTankCommandY
4      if self.tankX < 0:
5          self.tankX = 0
6      elif self.tankX >= self.worldWidth:
7          self.tankX = self.worldWidth - 1
8      if self.tankY < 0:
9          self.tankY = 0
10     elif self.tankY >= self.worldHeight:
11         self.tankY = self.worldHeight - 1
```

Lines 2-3 update the tank world coordinates using the command values. The rest of the method checks that these coordinates are inside the world and correct them if not the case. For instance, if the x coordinate is negative (line 4), we set it to 0. About coordinates range, remind that they are between 0 and the width or height minus 1.

2.6.5 What to remember

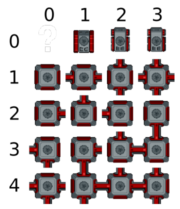
Pygame Surface blit() method: it copies a tile of a surface into another one:

```
surface.blit(tileset, location, rectangle)
```

- `surface`: the surface where we copy the tile;
- `tileset`: the surface with the tileset;
- `location`: the location `(x,y)` where we copy the tile;
- `rectangle`: a Pygame rectangle `(x,y,width,height)` that surrounds the tile in the tileset.

Tile and pixel coordinates: on screen, we draw elements at pixel coordinates. These coordinates are left-right and top-bottom and start with 0: the top-left coordinate is (0,0).

In a tileset, there are many tiles aligned on a regular grid. Each tile has a pixel coordinate, but we can also consider tile coordinates as the location in this grid:



With these coordinates, it is easier to locate tiles. For instance, the tank tile we used in the example is at tile coordinates (1,0). Then, to convert from tile coordinates `(tileGridX,tileGridY)` knowning that tile pixel size is `(tileWidth,tileHeight)` the calculation is:

```
tilePixelX = tileGridX * tileWidth
tilePixelY = tileGridY * tileHeight
```

Store parameters in variables: always use variables to store parameters. For instance, for tile width, we could use their raw value in computations:

```
tilePixelX = tileGridX * 64  # Immediate value: try to avoid!
```

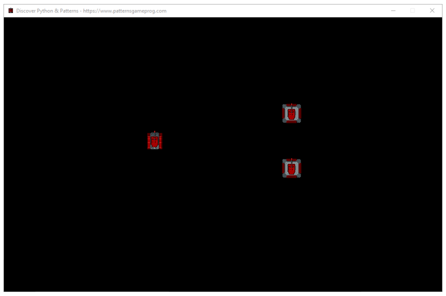
However, it will become complex to change this size later; furthermore, with a variable, the code is easier to understand:

```
tileWidth = 64  # Definition: once during construction
...
tilePixelX = tileGridX * tileWidth
```

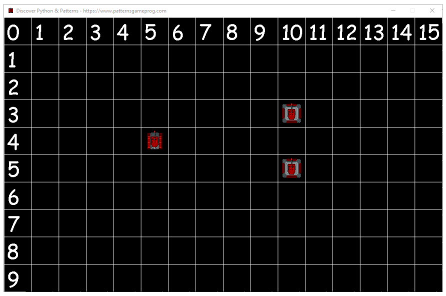
2.7 Add towers

2.7.1 World coordinates

We want to add two towers at locations (10,3) and (10,5). We use two sprites to draw them, one for the base and the other for the gun:



Locations are world coordinates, from left to right (x coordinate) and top to bottom (y coordinate). The lower coordinate value is 0, and the highest one is the width or height of the world minus one. So, with a world of 16 per 10 cells, the x coordinate goes from 0 to 15, and y from 0 to 9:



Looking at this screenshot, we can say that the tank is in cell (5,4). These coordinates are not related to pixel coordinates: whatever the size of tiles, the world locations are the same.

2.7.2 Add towers to the game state

We create new attributes in the `GameState` class:

- New attributes `tower1X`, `tower1Y`: the world coordinates of the first tower
- New attributes `tower2X`, `tower2Y`: the world location of the second tower

In the constructor of the `GameState` class, we define these new attributes:

```
def __init__(self):
    ...
    self.tower1X = 10
    self.tower1Y = 3
    self.tower2X = 10
    self.tower2Y = 5
```

2.7.3 Draw a unit with a gun

Drawing a tank or a tower with a gun is similar, except for location and base tile. Programmers don't like to type several times similar codes: the solution is to create a function or a method. Their arguments are the parameters that change from one result to the other. In our case, the location on-screen `(cellX,cellY)` and the coordinates `(tileX,tileY)` of the base tile are these parameters. As a result, we add the following method in the `UserInterface` class:

```
1  def drawUnit(self, cellX, cellY, baseTileX, baseTileY):
2      spriteX = cellX * self.tileWidth
3      spriteY = cellY * self.tileHeight
4      # Base
5      tileX = baseTileX * self.tileWidth
6      tileY = baseTileY * self.tileHeight
7      tileRect = Rect(tileX, tileY, self.tileWidth, self.tileHeight)
8      self.window.blit(self.unitsTileset, (spriteX, spriteY), tileRect)
9      # Gun
10     tileX = 4 * self.tileWidth
11     tileY = 1 * self.tileHeight
12     tileRect = Rect(tileX, tileY, self.tileWidth, self.tileHeight)
13     self.window.blit(self.unitsTileset, (spriteX, spriteY), tileRect)
```

Lines 2-3 create variables that contain the pixel location on the screen of the sprite. This location is the same for the base and the gun tile.

Lines 5-8 compute the rectangle of the base tile in the tileset and draw it. Lines 10-13 do the same for the gun, whose tile coordinates are (4,1).

The `render()` uses this method to render the three units:

```
def render(self):
    self.window.fill((0, 0, 0))

    gameState = self.gameState
    self.drawUnit(gameState.tower1X, gameState.tower1Y, 0, 1)
    self.drawUnit(gameState.tower2X, gameState.tower2Y, 0, 1)
    self.drawUnit(gameState.tankX, gameState.tankY, 1, 0)

    pygame.display.update()
```

Note that the coordinates of the tower tile are (0,1), and the coordinates of the tank tile are (1,0).

2.7.4 Collisions

If we add the previous changes, we see the two towers on the screen. However, if we move the tank, it can go over the towers. We can solve this with a new `update()` method of the `GameState` class:

```
1  def update(self, moveTankCommandX, moveTankCommandY):
2      newTankX = self.tankX + moveTankCommandX
3      newTankY = self.tankY + moveTankCommandY
4
5      if 0 <= newTankX < self.worldWidth \
6      and 0 <= newTankY < self.worldHeight \
7      and not(newTankX == self.tower1X and newTankY == self.tower1Y) \
8      and (newTankX != self.tower2X or newTankY != self.tower2Y):
9          self.tankX = newTankX
10         self.tankY = newTankY
```

Lines 2-3 store the target location in `newTankX`, `newTankY`. Then, we perform many checks and only update the tank location attributes if we succeed. This approach is more interesting than the previous one: if the new coordinates are invalid, we don't have to correct them; we just stop the update.

Lines 5-8 are a long `if` statement that does all the checks. Note the backslash \ at the end of the lines: splitting the lengthy statement into multiple lines makes it more readable. You can remove these backslashes and write all the conditions on a single line if it is unclear.

Line 5 checks that the new x coordinate is inside the world. It is a *chained comparison* and is equivalent to:

```
0 <= newTankX and newTankX < self.worldWidth
```

It ensures that the new x coordinate is 0 or more and strictly less than the world width (the last x coordinate is the world width minus one).

Line 6 is as line 5, but for the new y coordinate.

Lines 7-8 check that the new tank location does not equal the one of a tower. Both do the same operation but with a different syntax. Line 7 first compares the new tank location with the one of the first tower: `newTankX == self.tower1X and newTankY == self.tower1Y`. Then, it checks that it is not the case. We can split this check in the following way:

```
onTower1 = newTankX == self.tower1X and newTankY == self.tower1Y
notOnTower1 = not onTower1
if notOnTower1:  # The new location is NOT the one of tower 1
    ...
```

Line 8 does the same but for tower 2. It is a simplification of the previous syntax. According to boolean logic, `not(A and B) = A or B`. We apply this rule to `newTankX == self.tower2X and newTankY == self.tower2Y` and get the condition in line 8.

2.7.5 What to remember

World coordinates: the coordinates of an item in the world do not depend on the tile size or other rendering parameters. It is an important property, and if we keep it, we can design all the game logic without worrying about the rendering part.

Chained comparisons: we can check if a value is within a range with the syntax: `low < value < high` or `high > value > low`. We can switch strict comparison (<, >) with non-strict ones (<=, >=).

Boolean logic: we can simplify comparison using rules from the boolean logic:

- `not(A and B) = (not A) or (not B)`
- `not(A or B) = (not A) and (not B)`

Very long code lines: we can split a code line into multiple lines with the backslash \ operator. This expression:

```
result = expresion1 + expresion2 + expresion3 + expresion4
```

can also be written as:

```
result = expresion1 + expresion2 \
       + expresion3 + expresion4
```

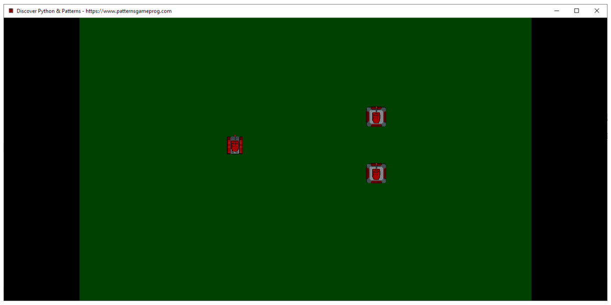
2.8 Rescaling and aspect ratio

2.8.1 Objective

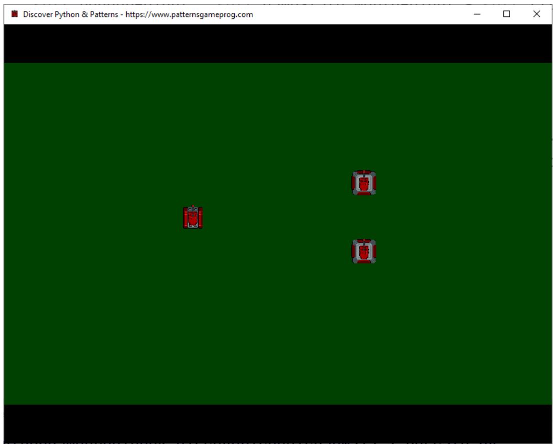
In the previous sections, we always create a window whose size depends on the world. Therefore, if the screen resolution is much lower or higher than its size, it leads to a poor user experience. We can solve this issue using the scale function inside Pygame.

Moreover, we want to keep the aspect ratio of the rendered world, which is its width divided by its height. If we rescale this rendering with a different aspect ratio, the world becomes stretched in one direction.

To takle this problem, we rescale the rendering with the maximum possible width or height (it depends on the shape of the window), and add black borders. For instance, if the aspect ratio of the window is smaller than the aspect ratio of the rendering, we add horizontal borders:


Discover Python & Patterns - https://www.patternsgameprog.com

In the other case, we add vertical borders:


Discover Python & Patterns - https://www.patternsgameprog.com

2.8.2 Initialization of the user interface

2.8.2.1 Define the rendering size

In the constructor of the `UserInterface` class, we create two new attributes `renderWidth` and `renderHeight` that define the size of the area to render:

```
def __init__(self):
    ...
    self.renderWidth = self.gameState.worldWidth * self.tileWidth
    self.renderHeight = self.gameState.worldHeight * self.tileHeight
    ...
```

We init these two values with the size of the world, as defined in the game state. This rendering size will always be the same, whatever the size of the window.

2.8.2.2 Define the window size

Still in the constructor of the `UserInterface` class, we compute a new window size:

```
    ...
    windowWidth = 1024
    windowHeight = (windowWidth*self.renderHeight)//self.renderWidth
    ...
```

We set the window width to an arbitrary value (here 1024, you can choose another one). Thus, whatever the size of the world, the window width has this value at startup. For the window height, we compute a value that leads to the same aspect ratio as the rendering.

Note the integer division operator `//`: the Pygame `set_mode()` expect integer values, so we only run integer operations (with the float division operator `/`, the result is always a float, even if the arguments are integers). Furthermore, we first multiply (because of the parenthesis) and then divide. If we run the opposite, the division would lose all precision, which would result in an error.

We can do some maths to check that the computation is right:

```
windowAspectRatio = windowWidth / windowHeight
 = windowWidth / (windowWidth * renderHeight / renderWidth)
 = (windowWidth / windowWidth) * (renderWidth / renderHeight)
 = renderAspectRatio
```

2.8.2.3 Create a resizable window

Also in the constructor of the `UserInterface` class, we create the main window:

```
    ...
    self.window = pygame.display.set_mode(
        (windowWidth, windowHeight),
        HWSURFACE | DOUBLEBUF | RESIZABLE
    )
    ...
```

Note the second argument of the `set_mode()` function: `HWSURFACE | DOUBLEBUF | RESIZABLE`. It is the combination of three options for window creation. We use the `|` operator to combine them: we want to enable all of them. The `HWSURFACE` and `DOUBLEBUF` options allow a faster blitting on the screen with video cards that support it (almost all cards today). With the `RESIZABLE` option, the user can resize the window.

Note that the full name of Pygame options starts with `pygame.constants.`. So, for instance, the full name of the resizable option is `pygame.constants.RESIZABLE`. Using an import at the beginning of the program, we can simplify this name:

```
from pygame.constants import RESIZABLE
```

2.8.3 Render the world

We create a new `renderWorld()` method that renders our world:

```
def renderWorld(self, surface):
    surface.fill((0, 64, 0))
    gameState = self.gameState
    self.drawUnit(surface, gameState.tower1X, gameState.tower1Y, 0, 1)
    self.drawUnit(surface, gameState.tower2X, gameState.tower2Y, 0, 1)
    self.drawUnit(surface, gameState.tankX, gameState.tankY, 1, 0)
```

As before, there are 3 units, a tank and two towers. Note the `surface` argument: this method can render in any surface.

2.8.4 Rescale and keep aspect ratio

We update the `render()` method of the `UserInterface` class:

```
1  def render(self):
2      # Render scene in a surface
3      renderWidth = self.renderWidth
4      renderHeight = self.renderHeight
5      renderSurface = Surface((renderWidth, renderHeight))
6      self.renderWorld(renderSurface)
7  
8      # Scale rendering to window size
9      windowWidth, windowHeight = self.window.get_size()
10     renderRatio = renderWidth / renderHeight
11     windowRatio = windowWidth / windowHeight
12     if windowRatio <= renderRatio:
13         rescaledWidth = windowWidth
14         rescaledHeight = int(windowWidth / renderRatio)
15         rescaledX = 0
16         rescaledY = (windowHeight - rescaledHeight) // 2
17     else:
18         rescaledWidth = int(windowHeight * renderRatio)
19         rescaledHeight = windowHeight
20         rescaledX = (windowWidth - rescaledWidth) // 2
21         rescaledY = 0
22 
23     # Scale the rendering to the window/screen size
24     rescaledSurface = pygame.transform.scale(
25         renderSurface, (rescaledWidth, rescaledHeight)
26     )
27     self.window.blit(rescaledSurface, (rescaledX, rescaledY))
28     pygame.display.update()
```

2.8.4.1 Render in a surface

Lines 3-6 render the world in a new surface. Line 5 creates a new Pygame surface with the rendering size. We can draw inside safely since it always has the same size. The second line renders the world in this surface. This rendering is in memory: we have to draw it in the window to see it on the screen.

2.8.4.2 Compute the aspect ratios

Lines 9-21 are the most important. They compute the size of our scene in the window without changing its aspect ratio. Otherwise, we could stretch the surface in one direction.

Line 9 gets the width and height of the window:

```
windowWidth, windowHeight = window.get_size()
```

The `get_size()` method of the `Surface` class returns a tuple of integers with the width and height of the surface. Note that we could also write:

```
windowWidth = window.get_width()
windowHeight = window.get_height()
```

Lines 10-11 compute the aspect ratios of the scene and window surfaces:

```
renderRatio = renderWidth / renderHeight
windowRatio = windowWidth / windowHeight
```

Note that the `/` division is a float division: `renderRatio` and `windowRatio` are float values (not integers). So then, depending on these ratios, the size of the rescaled rendering is different.

2.8.4.3 Compute the rescaled size

Lines 12-16 handle the case where the window aspect ratio is smaller or equal to the scene aspect ratio. It means that we can use the full window width but not the full height. Line 13 sets the rescaled width to the window width:

```
rescaledWidth = windowWidth
```

Line 14 computes a height that leads to a rescaled size with the aspect ratio of the rendering:

```
rescaledHeight = int(windowWidth / renderRatio)
```

Note the `int()` function that converts to integers. The scale function in the following expects integer values (not floats), so we make sure that it is the case.

Lines 15-16 compute the coordinates of the rescaled surface in the window in order to center it:

```
rescaledX = 0
rescaledY = (windowHeight - rescaledHeight) // 2
```

This time, we use the integer division operator `//`: `rescaledY` is an integer.

Lines 18-21 do the same, but for the case where we can use the full height of the window.

2.8.4.4 Scale and blit

Lines 24-26 use the `pygame.transform.scale()` function to rescale the rendering:

```
rescaledSurface = pygame.transform.scale(
    renderSurface, (rescaledWidth, rescaledHeight)
)
```

The first argument is the surface to rescale, and the second one is the target size.

Finally, line 27 blits the rescaled surface on the window/screen.

2.8.5 What to remember

Pygame Surface scale() function: it returns a rescaled surface:

```
rescaled = pygame.transform.scale(surface, size)
```

- `surface`: the surface to rescale;
- `size`: a tuple of integers (width,height) with the new size;
- `rescaled`: the output of the function.

Aspect ratio: it is the width divided by the height. If we change the aspect ratio of a Surface, it looks stretched in one direction. Most of the time, we cant to keep the aspect ratio of all rendered objects (sprites, scene, user interface components, etc.).

Integer and float division:

- Float division operator `/`: it always leads to a float, even if the arguments are integers;
- Integer division operator `//`: it always leads to an integer, even if the arguments are floats.

CHAPTER 3

Connect input, logic, and display

In this chapter, we continue the development of the tank game. We start to introduce more abstract concepts like lists and class inheritance. They allow us to create dynamic content and reduce the number of lines for the same result.

We also start to present the problems issued by synchronizing the game state and the user interface. Each one has its own pace, and we need to find solutions to connect each component efficiently.

You can find examples from this chapter here: `https://github.com/philippehenri-gosselin/discoverpythonpatterns/tree/master/chap03`. The names of the examples begin with the corresponding section. All the tileset images are in this folder.

3.1 Dynamic content with lists

In the previous program, we create one variable or class attribute for each unit. It works fine because we only have three units. This approach becomes very complex with more units, or if we need to add or remove units while the program is running.

We solve this issue with Python lists: we can store any number of objects in these containers. Furthermore, we can change their content anytime at runtime.

We first see a quick presentation of lists; then, we create a list to store the location of towers.

3.1.1 Lists

Create: creating a list with Python is easy; for instance, to create an empty list:

```
mylist = []
```

We can also create a list with an initial content:

```
mylist = [12, 25, 7]
```

Lists can contain any objects, and we can mix them:

```
mylist = ["Apple", 23, True]
```

Access: Python indexes items with a number from 0 to the size of the list minus 1:

```
mylist = [12, 25, 7]
print(mylist[0])      # print 12
print(mylist[1])      # print 25
print(mylist[2])      # print 7
```

We can see the corresponding value for each index of this list in the following table:

index	0	1	2
value	12	25	7

If we try to use an index higher or equal to the size of the list, then Python raises an exception:

```
mylist = [12, 25, 7]
print(mylist[3])      # raises an IndexError exception
```

We can also use negative indexes, in which case we access from the last item to the first one. If the index is too small (lower than the opposite of the size), then an exception is also raised:

```
mylist = [12, 25, 7]
print(mylist[-1])      # print 7
print(mylist[-2])      # print 25
print(mylist[-3])      # print 12
print(mylist[-4])      # raises an IndexError exception
```

Modification: we can modify any item in the list using the brackets:

```
mylist = [12, 25, 7]
mylist[1] = 3
print(mylist)       # print [12, 3, 7]
```

As for access, we can't use invalid indexes, e.g., values outside [-size,size-1].

We can add an item using the `append()` method:

```
mylist = [12, 25, 7]
mylist.append(3)
print(mylist)       # print [12, 25, 7, 3]
```

To remove an item, we can use the `del` statement:

```
mylist = [12, 25, 7]
del mylist[1]
print(mylist)       # print [12, 7]
```

Iterate through a list: we can access all items of a list using the `for` statement:

```
mylist = [12, 25, 7]
for item in mylist:
    print(item)
```

These lines print the following messages:

```
12
25
7
```

Note that the iteration is always from index 0 to the last one.

If we need to know the index of each item, we can use the `enumerate()` function:

```
mylist = [12, 25, 7]
for index, item in enumerate(mylist):
    print(index, ":", item)
```

These lines print the following messages:

```
0: 12
1: 25
2: 7
```

The iteration through a list is thanks to the Iterator pattern. We will see later how to create iterators. There are many other features around Python lists, and we show them throughout the book. But, for now, we have enough knowledge to improve our game.

3.1.2 Store the towers in a list

In the previous program, we store each tower position in a variable (constructor of the `GameState` class):

```
self.tower1X = 10
self.tower1Y = 3
self.tower2X = 10
self.tower2Y = 5
```

We can store the same locations using a list of tuples:

```
self.towersPos = [(10, 3), (10, 5)]
```

If we want to add another tower, we only need to add another item to this list, for instance, using the append() method:

```
self.towersPos.append((10,4))
```

This addition can be in the constructor or any other method of the GameState class. For instance, we can add one if the player presses a key. Consequently, the following implementations do not depend on the number of towers.

3.1.3 Handle collisions

We improve the update() method of the GameState class:

```
1  def update(self, moveTankCommandX, moveTankCommandY):
2      newTankX = self.tankX + moveTankCommandX
3      newTankY = self.tankY + moveTankCommandY
4
5      if not(0 <= newTankX < self.worldWidth) \
6      or not(0 <= newTankY < self.worldHeight):
7          return
8
9      for position in self.towersPos:
10         if newTankX == position[0] and newTankY == position[1]:
11             return
12
13     self.tankX = newTankX
14     self.tankY = newTankY
```

The key idea of this new implementation is to leave the function if a test fails. We use the return statement, which immediately leaves the current function (or method). If we meet any condition, the function execution stops, and Python ignores all the following lines.

Lines 5-7 check that the new tank position is not outside the world. If it is the case, we leave the method.

Lines 9-11 check that the new tank position is not a tower. It iterates through all tower positions. Inside the for block, the position variable contains a tower location. These locations are tuples of two integers: we can access their values with brackets as if it was a list of two elements.

For instance, we could write:

```
x = position[0]
y = position[1]
```

We can also pack these two lines in the following way:

```
x, y = position[0], position[1]
```

If we reach lines 13-14, it means we passed all checks, and we can update the tank position.

3.1.4 Render towers

The rendering of towers is as before, except we use the `for` statement to iterate through the tower list:

```
def renderWorld(self, surface):
    surface.fill((0, 64, 0))
    gameState = self.gameState
    for position in gameState.towersPos:
        self.drawUnit(surface, position[0], position[1], 0, 1)
    self.drawUnit(surface, gameState.tankX, gameState.tankY, 1, 0)
```

Lines 4-5 repeat a call to the `drawUnit()` method for each position in the list of tower positions.

3.1.5 What to remember

Lists:

- Create an empty list: `mylist = []`
- Create a list with content: `mylist = ["Apple", 23, True]`
- Get the size: `length = len(mylist)`
- Access an item: `age = mylist[1]`
- Change an item: `mylist[1] = 456`
- Add an item (at the end): `mylist.append(88)`
- Remove an item: `del mylist[1]`
- Iterate through a list:

```
    for item in list:
        ...
```

Tuples: they are like lists of a fixed size. We can access items in tuples as if it was a list, but we can never change them:

- Create a tuple: `mytuple = ("Apple", 23, True)`
- Access an item: `age = mytuple[1]`
- Change an item: impossible
- Add an item: impossible
- Remove an item: impossible
- Iterate through a tuple:

```
    for item in mytuple:
        ...
```

The enumerate() function: when used with `for`, it returns the index and the value of list items:

```
for index, value in enumerate(list):
    ...
```

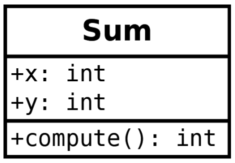
It also works with tuples.

3.2 Write less code with inheritance

In the current state of our game, we have two types of units: tank and tower. They have a different behavior but also share common properties. Right now, we duplicate code for these common properties. For instance, we have a code line that calls `drawUnit()` for the tank and one for towers. It is only one duplicated line of code now, but later, it will be a hundred lines. We can write these lines once using class inheritance.

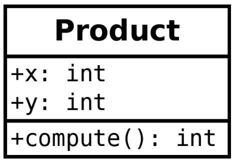
The next section introduces class inheritance. Then, the following one represents our units with this new feature.

3.2.1 Class inheritance



Classes with similar properties: a class allows us to “bundle” data in attributes and processes in methods. For instance, we create a class that stores two integers and has a single method that returns their sum. We can implement it in the following way:

```python
class Sum:
    def __init__(self,x,y):
        self.x = x
        self.y = y
    def compute(self):
        return self.x + self.y
```

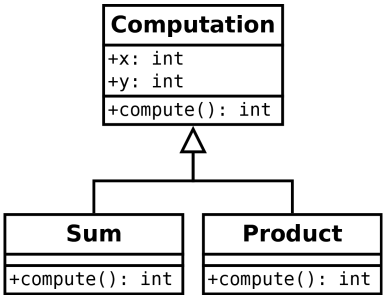


We also create a second class that stores two integers and computes their product. The graphical representation is the same, except for the name of the class. The implementation is also almost the same, except for the body of the `compute()` method:

```python
class Product:
    def __init__(self,x,y):
        self.x = x
        self.y = y
    def compute(self):
        return self.x * self.y
```

Gather similar properties: class inheritance allows gathering the shared code in a base class and creating specific implementations in child classes. For instance, for the case of computation classes:

```
class Computation:
    def __init__(self,x,y):
        self.x = x
        self.y = y
    def compute(self):
        pass

class Sum(Computation):
    def compute(self):
        return self.x + self.y

class Product(Computation):
    def compute(self):
        return self.x * self.y
```



As you can see, creating a new computation class (like `Sum` or `Product`) is quick and easy. We first create a class with a base class, with the following syntax:

```
class ChildClass(BaseClass):
    ... body ...
```

Then, we only need to implement specific methods. Note that it also works for the constructor (the `__init__()` method), meaning we can add new attributes in the child classes.

Method call: the cherry on the cake of class inheritance is that we can call a method in the base class without worrying about its implementation:

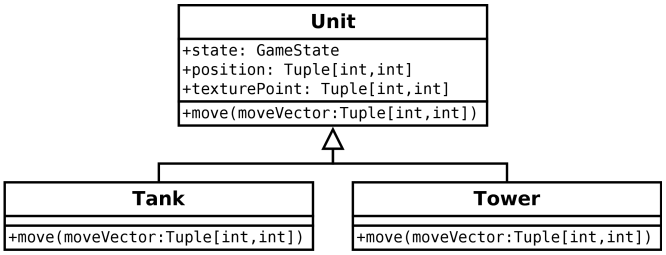
```
computations = [Sum(3, 2), Product(4, 5)]
for computation in computations:
    print(computation.compute())
```

In this example, the `for` loop only needs to know the `Computation` class, and more specifically, that this class has a `compute()` method.

We can add or remove new instances in the `computations` list, including new child classes of `Computation`, lines 2-3 do not need to be updated. Without class inheritance, we would have to add more code and edit this code for every new computation class. I hope that you see the incredible time-saving!

3.2.2 Unit classes

We use class inheritance to represent our units better:


Unit
+state: GameState
+position: Tuple[int,int]
+texturePoint: Tuple[int,int]
+move(moveVector:Tuple[int,int])
Tank
+move(moveVector:Tuple[int,int])
Tower
+move(moveVector:Tuple[int,int])

The `Unit` base class creates attributes: a reference to the game state (to access information like the world size), the location of the unit, and the tile in the tileset (for rendering):

```python
class Unit:
    def __init__(self, state, position, tile):
        self.state = state
        self.position = position
        self.tile = tile
    def move(self, moveVector):
        raise NotImplementedError()
```

The `move()` method raises a `NotImplementedError` exception. So, if we forget to implement this method in a child class, we will know it.

Tank move: the `Tank` child class implements the `move()` method. This implementation is very similar to what we were doing in the `GameState` class:

```python
1  def move(self, moveVector):
2      newPos = (
3          self.position[0] + moveVector[0],
4          self.position[1] + moveVector[1]
5      )
6      if not (0 <= newPos[0] < self.state.worldSize[0]) \
7      or not (0 <= newPos[1] < self.state.worldSize[0]):
8          return
9      for unit in self.state.units:
10         if newPos == unit.position:
11             return
12     self.position = newPos
```

Lines 2-5: the `position` attribute and the `moveVector` method argument are tuples of integers. The first value `variable[0]` is the x coordinate, and the second `variable[1]` is the y coordinate. We create a tuple `newPos` from these two ones.

Lines 6-8: we access the size of the world with the `state` attribute.

Lines 9-11: we still avoid any new location with a unit. This part is now more generic since it handles any units, like another tank player or new enemy type. Note the tuple comparison: the operator == returns `True` if two tuples have the same elements (same count, same values, same order).

Tower move: the `Tower` child class also implements the `move()` method but does nothing since towers can't move:

```python
class Tower(Unit):
    def move(self, moveVector):
        pass
```

The `pass` keyword means that we do nothing. Python requires it to create a block.

3.2.3 Game state

The implementation of the `GameState` class is simplified:

```python
1  class GameState:
2      def __init__(self):
3          self.worldSize = (16, 10)
4          self.units = [
5              Tank(self, (5, 4), (1, 0)),
6              Tower(self, (10, 3), (0, 1)),
7              Tower(self, (10, 5), (0, 1))
8          ]
9      def update(self, moveTankCommand):
10         for unit in self.units:
11             unit.move(moveTankCommand)
```

The constructor creates a list of units instead of locations (lines 4-8). It is more generic since we could add a unit of a different type with more specific properties. If the all-in-one syntax troubles you, we can write these lines in the following way:

```python
tank = Tank(self, (5, 4), (1, 0))
tower1 = Tower(self, (10, 3), (0, 1))
tower2 = Tower(self, (10, 5), (0, 1))
self.units = [tank, tower1, tower2]
```

The `update()` method calls the `move()` method of each unit (lines 10-11). The tank is the only one to move since the `move()` method of towers does nothing. Try to add a second one at a different initial location in the `units` list, and see the two tanks moving when you press the arrow keys.

Considering the Command pattern, the basic version we presented cannot handle more complex controls (like two players controlling two different tanks). Therefore, we need to introduce more concepts to show a better solution. Hence, it is the subject of a section in this chapter.

3.2.4 User Interface

The UserInterface class is also similar, except for the drawUnit() and renderWorld() methods. The first one has a unit argument from which we can get the position and tile of a unit. The second one only has a single call to drawUnit():

```
def renderWorld(self, surface):
    surface.fill((0, 64, 0))
    gameState = self.gameState
    for unit in gameState.units:
        self.drawUnit(surface, unit)
```

Note that we could also use class inheritance to specialize the rendering of each unit. However, we don't recommend it because it would mix game state and rendering.

3.2.5 What to remember

Create a child class: we create a child class with the following syntax:

```
class ChildClass(BaseClass):
    ... body ...
```

The child class has the same attributes and methods as the base class, except for the ones we redefine. We will use this feature a lot and see many examples in the following.

The pass keyword: it does nothing. We need it to create a block that does nothing

Tuple comparison: we can compare two tuples with == comparison operator. Two tuples are equal if there have the same elements (same count, same values, same order).

3.3 Render a background from a 2D array

We still have no background in our game: we add one in this section using 2D arrays. Before implementing this feature, we see some basics of 2D arrays.

3.3.1 2D arrays

We have already seen lists in Python, for instance, the creation of a list of three numbers:

```
list = [1, 2, 3]
```

An item of a list can be anything, including a list:

```
array2d = [[1, 2, 3], [4, 5, 6]]
```

Using this syntax, we create a 2D array with two rows of three items:

1	2	3
4	5	6

We can access each row of the array with one pair of brackets:

```
print(array2d[0]) # [1,2,3]
print(array2d[1]) # [4,5,6]
```

We can use a second pair of brackets to access the items of the 2D array:

```
print(array2d[0][1]) # 2
print(array2d[1][1]) # 5
```

More generally, we can access (or change) any item with the syntax `array2d[y][x]`, where `x` and `y` are the coordinates in the array.

We can iterate through 2D arrays using a double `for` statement:

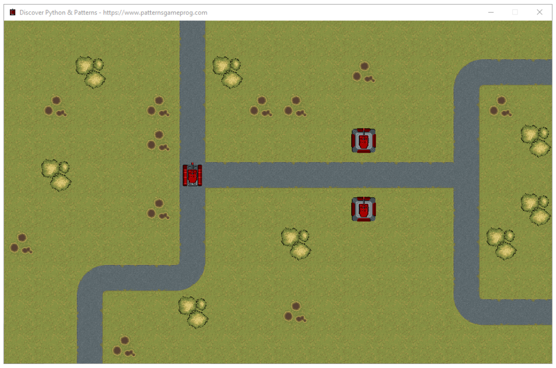
```
for y in range(2):
    for x in range(3):
        print(array2d[y][x], " ", end='')
    print()
```

The `end=''` argument in the `print()` call means that we don't want the printing of a carriage return. It leads to the following result:

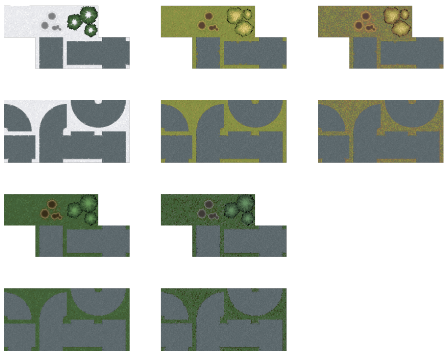
```
 1  2  3
 4  5  6
```

3.3.2 Tileset

Using a 2D array, we wish to add a background to our game:


Discover Python & Patterns - https://www.patternsgameprog.com

We also create a tileset using the sprites found here: https://zintoki.itch.io/ground-shaker, created by zintoki. We save this tileset in “ground.png”:



In the constructor of the `UserInterface` class, we load this tileset in a new attribute `groundTileset`:

```python
def __init__(self):
    ...
    self.groundTileset = pygame.image.load("ground.png")
    ...
```

3.3.3 Create the 2D array

In the constructor of the `GameState` class, we create a new attribute `ground` with the coordinates of tiles in the ground tileset. For instance, `(5,1)` are the coordinates of the green grass tile. The definition of this attribute looks like this (we don’t show all lines):

```
self.ground = [
    [(5, 1), (5, 1), (5, 1), (5, 1), (5, 1), (6, 2), (5, 1), (5, 1), (5, 1),
 (5, 1), (5, 1), (5, 1), (5, 1), (5, 1), (5, 1), (5, 1)],
    [(5, 1), ...
    ...
]
```

Creating such a 2D array directly with Python code is not handy; we will see later how to make them with software like Tiled.

3.3.4 Render the tiles from the 2D array

We create a new method `drawGround()` in the `UserInterface` class, which draws a single tile. It works as before (like in the `drawUnit()` method):

```
def drawGround(self, surface, position, tile):
    spriteX = position[0] * self.tileWidth
    spriteY = position[1] * self.tileHeight
    tileX = tile[0] * self.tileWidth
    tileY = tile[1] * self.tileHeight
    tileRect = Rect(tileX, tileY, self.tileWidth, self.tileHeight)
    surface.blit(self.groundTileset, (spriteX, spriteY), tileRect)
```

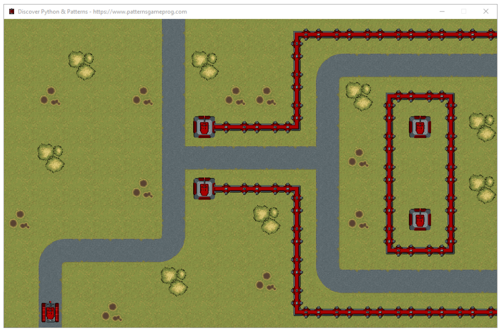
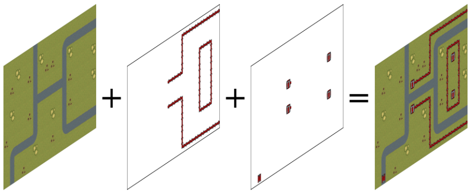
In the `renderWorld()` method of the `UserInterface` class, we add the rendering of all tiles of the 2D array. It uses a double `for` statement:

```
def renderWorld(self, surface):
    ...
    for y in range(gameState.worldSize[1]):
        for x in range(gameState.worldSize[0]):
            self.drawGround(surface, (x, y), gameState.ground[y][x])
    ...
```

3.3.5 What to remember

2D arrays with lists:

- Create: `array2d = [[a, b, c], [e, f, g], ...]`
- Get a value at (x,y): `value = array2d[y][x]`
- Set a value at (x,y): `array2d[y][x] = value`
- Resize: no simple operation


Discover Python & Patterns - https://www.patternsgameprog.com


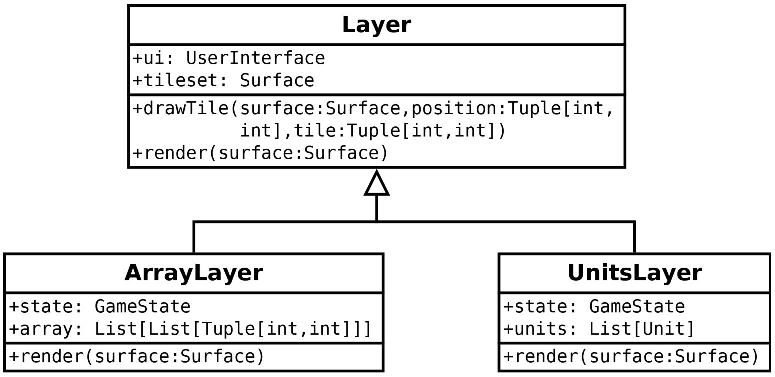
Rendering with three layers: ground, walls, and units.

3.4 Render with layers

We got a background now, but we can get a better one with layers. We use class inheritance to add these new layers of different kinds easily.

3.4.1 Layers with a class hierarchy

We want to get the result in the figure above, where we add walls on top of the background. So we use three layers: one for the ground, one for the walls, and the last for the tank and towers. We could get such a result by copying and pasting the code that renders the background. However, we can do much better with a class hierarchy: the base class contains all the shared features, and each child class the code specific to each case:



The `Layer` class is the base class that contains shared data and functionalities:

- A `ui` reference to the `UserInterface` to access data like the size of tiles;
- A `tileset` Pygame surface that contains the image tileset;
- A `renderTile()` method that draws a tile from the tileset anywhere on a surface;

- A `render()` method that we can call to render the layer in a surface. Child classes must implement this method.

The `ArrayLayer` child class renders layers based on a 2D array, like the background we added in the previous post. It contains the following members:

- A `state` reference to a `GameState` to access data like the size of the world. Note that it is not in the base class because we could create layers that need no access to a game state;
- A `array` reference to a 2D array of tuples, like the one we created in the `GameState` class;
- A `render()` method that renders the layer using the 2D array.

The `UnitsLayer` child class is similar to the `ArrayLayer` class, except it references a list of `Unit` class instances.

3.4.2 The Layer base class

```python
class Layer:
    def __init__(self, ui, imageFile):
        self.ui = ui
        self.tileset = pygame.image.load(imageFile)

    def drawTile(self, surface, position, tile):
        tileWidth = self.ui.tileWidth
        tileHeight = self.ui.tileHeight
        spriteX = position[0] * tileWidth
        spriteY = position[1] * tileHeight
        tileX = tile[0] * tileWidth
        tileY = tile[1] * tileHeight
        tileRect = Rect(tileX, tileY, tileWidth, tileHeight)
        surface.blit(self.tileset, (spriteX, spriteY), tileRect)

    def render(self, surface):
        raise NotImplementedError()
```

The constructor creates the two attributes. We load an image file into the `tileset` attribute: when using this constructor, we only need to give the name of the image file.

The `drawTile()` method is similar to previous methods like `drawGround()` or `drawUnit()`. Now, it is the only location in the program where we see such a code!

The `render()` method raises an exception: child classes must implement it.

3.4.3 The ArrayLayer class

```
class ArrayLayer(Layer):
    def __init__(self, ui, imageFile, gameState, array):
        super().__init__(ui, imageFile)
        self.state = gameState
        self.array = array

    def render(self, surface):
        for y in range(self.state.worldSize[1]):
            for x in range(self.state.worldSize[0]):
                tile = self.array[y][x]
                if tile is not None:
                    self.drawTile(surface, (x, y), tile)
```

The constructor expects four arguments: the two arguments from the base class constructor (`ui` and `imageFile`) and the two arguments for the specific attributes of this class (`gameState` and `array`).

Note the first line of the constructor:

```
super().__init__(ui,imageFile)
```

It is the syntax to call a method from the base class (or "super" class). We can do the same for any other base method, for instance, `super().render(surface)` calls the `render()` method of the base class:

The `render ()` method is similar to the block that renders the ground in the previous `renderWorld()` method of the `UserInterface` class. However, it is no longer limited to the `ground` 2D array and can render any 2D array. In addition, it handles empty cells in the array: if there is a `None` value, which means "nothing" or "empty" in Python, we render no tile.

3.4.4 The UnitsLayer class

This class is similar to `ArrayLayer`, except we work with a `Unit` list instead of a 2D array. The `render()` method iterate through all units, and call twice the `drawTile()` method: one time to draw the unit tile (base tank or tower) and another time to draw the gun:

```
def render(self, surface):
    for unit in self.units:
        self.drawTile(surface, unit.position, unit.tile)
        self.drawTile(surface, unit.position, (4, 1))
```

3.4.5 The UserInterface class

In the constructor of the `UserInterface` class, we add a list of layers. It is like we did for the `units` in the `GameState` class: we can create a list of any size, and then the class of each item defines its behavior:

```
self.layers = [
    ArrayLayer(self, "ground.png", gameState, gameState.ground),
    ArrayLayer(self, "walls.png", gameState, gameState.walls),
    UnitsLayer(self, "units.png", gameState, gameState.units)
]
```

We no more need the `drawGround()` and `drawUnit()` methods for the rendering. To render the world, we only need the following `renderWorld()` method:

```
def renderWorld(self, surface):
    for layer in self.layers:
        layer.render(surface)
```

Pay attention to the `for` statement: it is similar to what we did for the units in the `GameState` class. For each layer in our layers list, we ask for a rendering. At this point, we don't need to worry about the type of layers in the list. We could change them, create or delete new layer classes; these lines still work.

3.4.6 Game state update

We update the `move()` method of the `Tank` class to make sure that the tank can't walk on walls:

```
def move(self, moveVector):
    ...
    if self.state.walls[newPos[1]][newPos[0]] is not None:
        return
    ...
```

This new line checks if there is a wall at the new tank position. If it is the case, we leave the method, which cancels the move.

3.4.7 What to remember

Save code lines with class hierarchy: we have already seen this technique in the previous sections, but it is worth repeating. Good programmers don't like to type many time similar programs: it takes time and is dangerous when we need to do updates. Instead, we can gather shared code with functions; it is even better with classes since it collects data in attributes.

Abstraction with class hierarchy: in the game update and the rendering, we trigger processing of all items from a list (units in the first case, layers in the second):

```
for item in list:
    item.doSomething()
```

It is possible because each item implements the same method (here doSomething()). We don't know what it exactly does, and we don't need to.

Call a base method: the super() function calls a base method from a child method. The method we call does not need to be the same; we can call any method from the base class:

```
def childMethod():
    super().baseMethod()
```

3.5 Tile rotation and mouse handling

We continue implementing our game and want to orient the unit guns. The tank targets the mouse position, and the tower the tank. Before considering the mouse handling, we need to rotate unit guns correctly.

3.5.1 Render a rotated tile

In the `Layer` class, we add a new `angle` argument to the `drawTile()`, which renders a tile from a tileset. If this argument is not `None`, we rotate the tile:

```
1  def drawTile(self, surface, position, tileCoords, angle=None):
2      tileWidth = self.ui.tileWidth
3      tileHeight = self.ui.tileHeight
4      spriteX = position[0] * tileWidth
5      spriteY = position[1] * tileHeight
6      tileX = tileCoords[0] * tileWidth
7      tileY = tileCoords[1] * tileHeight
8      tileRect = Rect(tileX, tileY, tileWidth, tileHeight)
9      if angle is None:
10         surface.blit(self.tileset, (spriteX, spriteY), tileRect)
11     else:
12         tile = pygame.Surface((tileWidth, tileHeight), SRCALPHA)
13         tile.blit(self.tileset, (0, 0), tileRect)
14         rotatedTile = pygame.transform.rotate(tile, angle)
15         spriteX -= (rotatedTile.get_width() - tile.get_width())//2
16         spriteY -= (rotatedTile.get_height() - tile.get_height())//2
17         surface.blit(rotatedTile, (spriteX, spriteY))
```

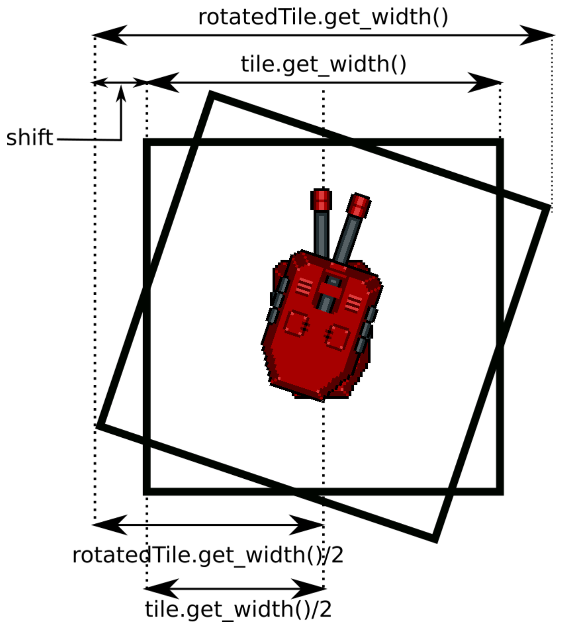
The `angle` argument in the method declaration (line 1) has a default value set to `None`. Therefore, if we call the method without this last argument, `angle` gets this default value. The first lines (2-8) are unchanged: we compute the location on the surface and the rectangle of the tile in the texture tileset. Then, if `angle` is `None` (line 9), we render as before (line 10).

Lines 12-17 rotate and render the tile.

Lines 12-13 extract the tile from the tileset. It is a Pygame surface with the size of a tile, e.g. 64 per 64 pixels. We create a new pygame surface with an alpha channel, to keep transparency (line 12). The `SRCALPHA` value from `pygame.constants` enables this feature.

Line 14 rotates the tile using the Pygame `scale()` function, and store the result in `rotatedTile`. This new surface can be larger than the tile. The figure can help you better understand why (we added black borders to see the edges of the tile).

Consequently, since we draw from the top left corner of the tile, we have to draw the sprite with a small shift. We compute this shift in lines 15-16 and update `spriteX,spriteY` coordinates accordingly. The following figure can help you understand how we compute the shift:



3.5.2 Weapon target

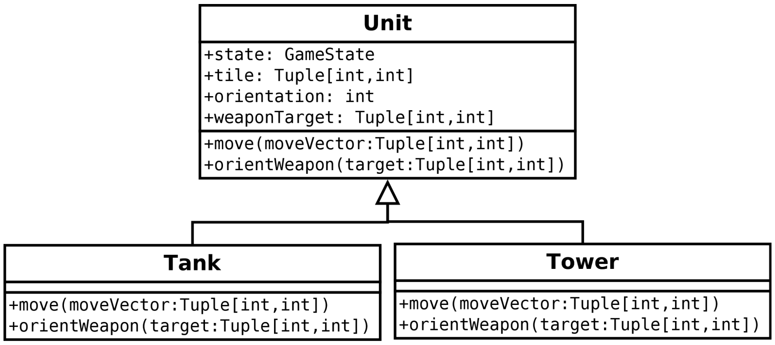
In the `Unit` class, we add a new `weaponTarget` attribute with the cell's coordinates targeted by the unit weapon.

The `render()` method of the `UnitsLayer` class uses this attribute. It is as before, except that we compute the angle between the current unit position and the targeted cell (lines 5-7):

```python
1 def render(self, surface):
2     for unit in self.units:
3         self.drawTile(surface, unit.position,
4                       unit.tile, unit.orientation)
5         dirX = unit.weaponTarget[0] - unit.position[0]
6         dirY = unit.weaponTarget[1] - unit.position[1]
7         angle = math.atan2(-dirX, -dirY) * 180 / math.pi
8         self.drawTile(surface, unit.position, (4, 1), angle)
```

3.5.3 Target selection

We can now render a unit with a weapon oriented towards any cell. We need to choose which one is targeted by each unit. We first create a new `orientWeapon()` method in the `Unit` class hierarchy:


Unit
+state: GameState
+tile: Tuple[int,int]
+orientation: int
+weaponTarget: Tuple[int,int]
+move(moveVector:Tuple[int,int])
+orientWeapon(target:Tuple[int,int])
Tank
+move(moveVector:Tuple[int,int])
+orientWeapon(target:Tuple[int,int])
Tower
+move(moveVector:Tuple[int,int])
+orientWeapon(target:Tuple[int,int])

The principle is as for the `move()` method: the implementation in the base class raises a `NotImplementedError` exception, and implementations in child classes are specific.

For the tank, we copy the `target` method argument to the `weaponTarget` attribute. Later, we use the mouse coordinates to define `target`, and the tank weapon always targets the mouse cursor:

```python
class Tank(Unit):
    ...
    def orientWeapon(self, target):
        self.weaponTarget = target
```

We ignore the method argument for towers and copy the tank position (the first unit in the list of units). This way, all towers always target the tank:

```python
class Tower(Unit):
    ...
    def orientWeapon(self, target):
        self.weaponTarget = self.state.units[0].position
```

From a design point of view, this is not good: we assume that there is always at least one unit in the game state and that this unit is the tank. We will solve this issue later.

3.5.4 Mouse handling and Command pattern

All that remains is using the mouse coordinates to orient the tank weapon. As before, we use a basic implementation of the Command pattern. So we need:

- To create a variable that stores the cell targeted by the mouse;
- To compute the target cell coordinates;
- To transmit this value to all units.

Following this recipe, we create a new targetCommand attribute in the UserInterface class. It contains the coordinates targetted by the mouse.

The processInput() method of the UserInterface class computes the target cell coordinates, and store them in the targetCommand attribute:

```
def processInput(self):
    ...
    mousePos = pygame.mouse.get_pos()
    mouseX = (mousePos[0] - self.rescaledX) / self.rescaledScaleX
    mouseY = (mousePos[1] - self.rescaledY) / self.rescaledScaleY
    self.targetCommand = (
        mouseX / self.tileWidth - 0.5,
        mouseY / self.tileHeight - 0.5
    )
```

The pygame.mouse.get_pos() function returns the mouse pixel coordinates in the window. We must invert the scaling transform since we rescale the world to fit the window. During the rescaling (in the render() method), we save in new attributes rescaledX,rescaledY the shift and in rescaledScaleX,rescaledScaleY the scale factors.

We divide these coordinates by the size of a cell to get cell coordinates. Then, we subtract 0.5 to target the center of the cell. We compute cell coordinates to be independent of the pixel size of tiles. Furthermore, these coordinates are float numbers (numbers with decimals) to target any screen pixel.

The update() method of the UserInterface class transmits all commands (move and target) to the game state:

```
def update(self):
    self.gameState.update(self.moveTankCommand, self.targetCommand)
```

Finally, the update() method of the GameState class gives the moveTankCommand to the move() method of all units, and gives the targetCommand to the orientWeapon() method of all units. Then, each unit can use (or don't use) these commands:

```
def update(self, moveTankCommand, targetCommand):
    for unit in self.units:
        unit.move(moveTankCommand)
        unit.orientWeapon(targetCommand)
```

Only the tank uses the commands in our current implementation, and the towers ignore them. You can try to change it; for instance, let all towers target the mouse cursor or one of the towers. You only need to change the `orientWeapon()` method of the `Tower` class to get such results.

In the final program, we also add an `orientation` attribute to units, so the base of each unit can be oriented. We use this attribute to orient the tank base towards its current move. Finally, in the `move()` method of the `Tank` class, we update this attribute depending on the move command.

3.5.5 What to remember

Game state independence: programmer's life is easier when the game data and logic are independent of the UI parameters, like the screen's resolution. Some games depend on their rendering in their logic, like high-speed fighting games (ex: Street Fighters II). For other games, we better remove this dependence; otherwise, we risk a lot of headaches.

In the case of mouse coordinates, we could create a mouse command with pixel screen coordinates. Everything works fine as long as the size of the tiles is the same. The day we change this, the game logic fails, and strange behaviors appear. In the meantime, we need pixel precision.

The trick we use to solve this issue is to use float numbers. Their round part corresponds to cell coordinates (independent of tile size) and their decimal to pixel coordinates. For instance, coordinates (3.5,2.1875) correspond to cell (3,2) and pixel (32,12) in the cell, if tile size is (64,64). The computation is 32 = 0.5 * 64 and 12 = 0.1875 * 64.

Method default argument value: We can define a default value to function and method arguments:

```python
def f (a, b=2, c="hello"):
    ...
f(10)  # same as f(10,2,"hello")
```

To define a default value for an argument, all the following ones must also have a default value. For instance, `def f(a=2,b)` is impossible.

Rotate a surface: we can rotate a surface with the `rotate()` Pygame function:

```python
rotated = pygame.transform.rotate(surface, angle)
```

`surface` is any surface, and `angle` is an integer value (degrees, from 0 to 360). The function returns a new surface whose size can be larger.

Create a surface with transparency: add the `SRCALPHA` flag when creating a new surface:

```python
from pygame.constants import SRCALPHA
suface = pygame.Surface(size, SRCALPHA)
```

Get mouse position: the `pygame.mouse.get_pos()` function returns the pixel coordinates of the mouse cursor, relatively to the window/screen (coordinates (0,0) is top-left corner):

```
mouseX, mouseY = pygame.mouse.get_pos()
```

3.6 Better commands

Since we know class inheritance, we can implement the Command pattern better. Moreover, this approach eases the management of commands and introduces exciting features.

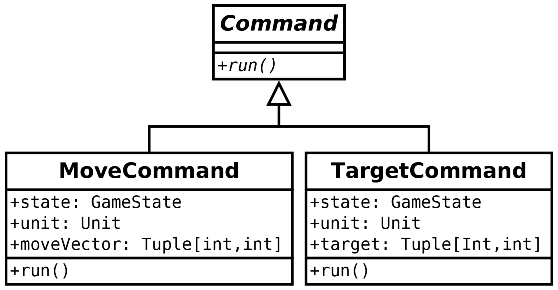
3.6.1 The Command pattern

The Command pattern has two main steps: store the information required for updating data, and execute these updates later, eventually in a different order.

In the previous programs, we implement this recipe using a variable to store our commands. For instance, the `moveTankCommand` attribute of the `UserInterface` class memorizes the direction in which the tank should move. Then, during the game update, we use this attribute to move the tank.

Storing in a single attribute is limited, and we can do much better if we store the commands in a class instance. Furthermore, we can also embed the update process in this class rather than putting it somewhere in the main game state class.

Using these ideas, we implement the Command pattern for our two current commands (move and target) using the following class hierarchy:



The `Command` class is the base class. It only contains a `run()` method that executes the tasks of the command instance. The `MoveCommand` is a child class of `Command` and stores all the information we need to move a unit. Note that we extend the command to control any unit (tanks or towers), which leads to funny gameplay for free, like controlling several tanks or towers. The `TargetCommand` is also a child class of `Command` and handles the orientation of a weapon towards a unit.

3.6.2 Changes to the `Unit` class hierarchy

In the previous sections, we created a `Unit` class hierarchy (the `Tank` and `Tower` classes) to represent all units and handle their data update.

This approach mixes the data and its update; many software architectures use this recipe because it is easy to use and understand. However, it has several flaws and fewer features than the one we propose to use in this section. The main issue is that an update focuses on a single object (the class instance). For instance, we implemented the tank's move in the `Tank` class; if we want to move other units with the same code, we must duplicate it.

As the game becomes more and more complex, this case is less likely to happen. When we have to update several game items resulting from one command, we must choose one of the classes involved. For instance, when one unit destroys another, which class should implement the destruction? The destroying or the destroyed? It is a simple case; some commands can update the data of many items or parts of the world.

There are no such flaws with the approach we propose to follow here. The state classes store data on one side, and the command classes update data on the other side. As a result, it follows the most crucial rule of software design: divide problems into sub-problems!

Considering our `Unit` class hierarchy, we no more need it. A single `Unit` class can store the data of any unit:

```python
class Unit:
    def __init__(self, state, position, tile):
        self.state = state
        self.position = position
        self.tile = tile
        self.orientation = 0
        self.weaponTarget = (0, 0)
```

We no more need a class hierarchy because unit types don't require any new attribute or method.

3.6.3 Command classes

Command class: the implementation of the base class is straightforward. The `run()` method only raises an exception, in case we forgot to implement this method in a child class:

```python
class Command():
    def run(self):
        raise NotImplementedError()
```

TargetCommand class: this class updates the `weaponTarget` of the `unit` attribute:

```python
class TargetCommand(Command):
    def __init__(self, state, unit, target):
        self.state = state
        self.unit = unit
        self.target = target
    def run(self):
        self.unit.weaponTarget = self.target
```

It is similar to what we did in the `orientWeapon()` methods of the previous `Tank` and `Unit` classes. Note that there is no more binding to the first unit of the unit lists.

MoveCommand class: we define tree attributes `state`, `unit` and `moveVector`, and update the previous `update()` method of the `Tank` class into the `run()` method. We change how we reach data; for instance, `self.orientation` becomes `self.unit.orientation`.

3.6.4 Store the commands

We store all commands in a list, in a new attribute of the `UserInterface` class. This way, we can have any number of commands:

```
class UserInterface():
    def __init__(self):
        ...
        self.commands = []
        self.playerUnit = self.gameState.units[0]
```

We also create a new `playerUnit` attribute. It references the unit controlled by the player. Remember that, except for numbers, all Python variables contain a reference to an object. As a result, `playerUnit` "points" to the same unit as the first item of `units` in the game state. So, if we do something to `playerUnit` (we should say "to what `playerUnit` refers"), it also updates `gameState.units[0]`, and vise-versa.

3.6.5 Create the commands

The creation of commands is still in the `processInput()` method of the `UserInterface` class. We create child classes of `Command` rather than setting a single attribute:

```
def processInput(self):
    state = self.gameState
    playerUnit = self.playerUnit
    moveX, moveY = 0, 0
    for event in pygame.event.get():
        ...handle quit and update moveX,moveY...
    if moveX != 0 or moveY != 0:
        command = MoveCommand(state, playerUnit, (moveX, moveY))
        self.commands.append(command)

```

```
11        mousePos = pygame.mouse.get_pos()
12        mouseX = (mousePos[0] - self.rescaledX) / self.rescaledScaleX
13        mouseY = (mousePos[1] - self.rescaledY) / self.rescaledScaleY
14        targetCell = (
15            mouseX / self.tileWidth - 0.5,
16            mouseY / self.tileHeight - 0.5
17        )
18        command = TargetCommand(state, playerUnit, targetCell)
19        self.commands.append(command)
20
21        for unit in state.units:
22            if unit != playerUnit:
23                command = TargetCommand(state, unit, playerUnit.position)
24                self.commands.append(command)
```

Lines 4-6 compute the move vector using the Pygame keyboard events and store it in `moveX,moveY`. Lines 7-9 create an instance of the `MoveCommand` class and add it to the list of commands.

Lines 11-17 compute the cell targeted by the mouse cursor as before. Then, it creates an instance of the `TargetCommand` class and adds it to the list of commands (lines 18-19). The controlled unit is the one referenced by `playerUnit`.

Lines 12-24 create a target command for all units except the one controlled by the player. It is to get the same behavior as in the previous section, where all towers target the tank.

3.6.6 Commands are not only for player items

A legitimate question is: why update the non-playing game items with commands? It looks much more natural to directly update the towers in their respective classes. A quick answer is: what if we want to control another unit? With the proposed scheme, we only need to change the `playerUnit` attribute, and the player can control a tower, and all other units, including the tank, will point to this tower.

Using commands to control any game state update is even more powerful. Following this approach, we have complete control of the game updates. Thanks to the commands list, we know the order of command execution. We can also change this order afterward. For instance, it can be interesting to move the tank first, even if the player does not control it. We can log these commands and better understand why we get one behavior instead of another. Admittedly, it is not mandatory for a small game like the one we are creating. But think about a more complex game, like an MMORPG where 10,000 players send a dozen commands each second. It is infinitely easier to find a problem when fully controlling the execution flow.

If the motivation behind using this pattern is not clear, don't worry. Please trust the experimented developers, and it will save you valuable time!

3.6.7 Update the game state

Update the game state is simple: we only have to run all the commands. This task is in the `update()` method of the `UserInterface` class, and replace the call to the previous `update()` method of the `GameState` class (which is no more required):

```
1 class UserInterface():
2     ...
3     def update(self):
4         for command in self.commands:
5             command.run()
6         self.commands.clear()
```

Once we have executed these commands (lines 4-5), we must clear the commands list (line 6). Otherwise, they will repeat endlessly!

3.6.8 What to remember

The Command pattern with a class hierarchy: Create a base class with a `run()` or `execute()` method. Then, each child class stores all they need in their attributes and implements this method to perform their task.

Generalist commands: We don't have to dedicate commands to specific game items. Instead, we can create commands that work for many items (when possible). The move command we created is a good example: it can work for a tank or a tower.

Passive commands: The player is not the only one to create commands. The game logic itself can create commands to update the game. For instance, a command can update the gold of each town in a strategy game. This way, we control the execution flow with the command list. Later, we will see that we can organize commands with a priority level.

When to create a class hierarchy: Most of the time, we need a class hierarchy when we need to introduce new attributes or methods for a new data type.

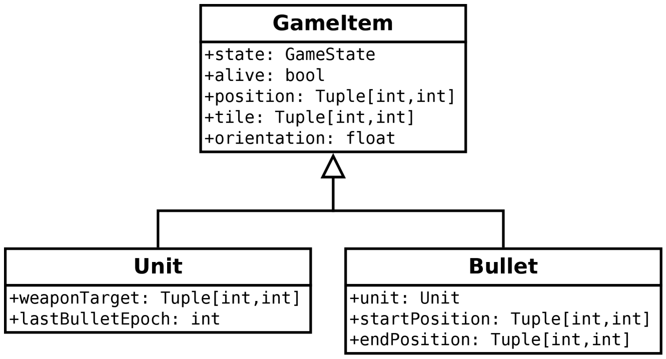
3.7 Items life cycle

It is time to shoot with our tank and destroy the towers! We have all we need: a state to represent bullets, commands to update them, layers for the rendering, and a UI to manage all of these.

Contrary to other units, bullets do not exist at the beginning of the program. Instead, they appear when the player clicks and disappear when they hit a unit or after a while. So, we have to find a way to add and remove these new items without introducing new issues.

3.7.1 Game items

Bullets are too different from the tank and towers, so we need to (re)introduce a class hierarchy, with a child class for units and a child class for bullets:



The `GameItem` base class contains the shared attributes: a reference to the game state, if the item is alive, its cell position, its tile coordinates, and its orientation.

The `Unit` child class contains the cell's coordinates targeted by its weapon. We also add the last time the unit fired: it limits the number of bullets a unit can shoot. We call the game time “epochs”; we detail this in the next sections.

The `Bullet` child class contains all that we need to manage a bullet:

- A reference to the unit that shot it (`unit`);
- The starting cell coordinates (`startPosition`). We initialize it with the targeting unit position;
- The final cell coordinates (`endPosition`). We initialize it with the targetted unit position.

3.7.2 Game state and game epochs

In the `GameState` class we add the following new attributes:

- `bullets`: a list of current bullets;

- `bulletSpeed`: a float number that defines the speed of bullets;
- `bulletRange`: a float number that defines the range of bullets;
- `bulletDelay`: a float number that defines the minimum number of game epochs between two shots;
- `epochs`: current game epoch.

We don't consider time to handle game time (in seconds or minutes, for instance). Since each computer can run at a different speed, depending on its power, the game may not run at the same speed on every device. A usual approach considers game epochs, which we increase at each game update.

If the computer can run the game at maximum speed, for instance, 60 game updates per second, we get perfect synchronization between game epochs and time. In this case, a game epoch always lasts about 16 milliseconds. However, there are no guarantees that every computer can run updates 60 times per second.

Whatever the computer's speed, the gameplay must always be the same. For example, considering the bullets, we don't want a player that shoots more bullets because he/she has a faster computer! Working with game epochs is one of the best ways to achieve this since it no longer depends on computing power. In the worst case, the display gets slower, or we render fewer frames per second.

3.7.3 Bullets commands

The creation and destruction of items during the game can lead to problems. For example, we store bullets in a list (`bullets` attribute in the game state), so we must ensure that this list does not change as long as we reference an item somewhere in the code. Otherwise, we could update a bullet no longer in the list or add a bullet to the list while another process iterates through it.

The Command pattern can solve this issue since we can control the processing order. So, to update bullets, we create three commands:

- `ShootCommand` to create a new bullet;
- `MoveBulletCommand` to move a bullet and act accordingly;
- `DeleteDestroyedCommand` to delete "dead" items (including bullets).

We first schedule `ShootCommand` (add to list), then `MoveBulletCommand` (update in the list), and finally `DeleteDestroyedCommand` (remove from the list). This way, we never mix up list processing.

3.7.3.1 ShootCommand

The `ShootCommnand` class stores a reference to the game state and to a unit that shoots. Its `run()` method adds a new bullet to the list in the state if it is possible:

```python
def run(self):
    unit = self.unit
```

```
3        if not unit.alive:
4            return
5        state = self.state
6        if state.epoch - unit.lastBulletEpoch < state.bulletDelay:
7            return
8        unit.lastBulletEpoch = state.epoch
9        state.bullets.append(Bullet(state, unit))
```

Lines 6-7 cancel the addition if the unit recently shot. We compute the difference between the current state epoch and the epoch when the unit fired. If this difference is less than the `bulletDelay` defined in the game state, we don't add a new bullet.

3.7.3.2 MoveBulletCommand

The `MoveBulletCommand` class stores a reference to the game state and a bullet and handles its movement. The whole process is complex and specific to the game; we don't describe it all. However, we can point out a few tricks to achieve more readable code. For instance, we need to compute the new position $\mathbf{p}_{new}$ of the bullet, which is

$$\mathbf{d} = \frac{\mathbf{p}_{end} - \mathbf{p}_{start}}{||\mathbf{p}_{end} - \mathbf{p}_{start}||}, \quad \mathbf{p}_{new} = \mathbf{p}_{now} + s\mathbf{d}$$

with $\mathbf{p}_{now}$ the current position, $\mathbf{p}_{start}$ the first position, $\mathbf{p}_{end}$ the last position, $\mathbf{d}$ the direction the bullet is heading to, and s the bullet speed.

This kind of processing is easier to implement (and later to read) if we implement new functions. For instance, the direction computation is a vector subtraction followed by normalization. Using the following functions:

```python
def vectorSub(a, b):
    return a[0] - b[0], a[1] - b[1]
def vectorNorm(a):
    return math.sqrt(a[0]**2 + a[1]**2)
def vectorNormalize(a):
    norm = vectorNorm(a)
    if norm < 1e-4:
        return 0, 0
    return a[0] / norm, a[1] / norm
```

We can compute the direction:

```python
direction = vectorSub(bullet.endPosition, bullet.startPosition)
direction = vectorNormalize(direction)
```

This code is easy to write: each function is straightforward, and the final computation speaks for itself. If we don't follow this approach, the code looks like this:

```
direction = bullet.endPosition[0] - bullet.startPosition[0], \
            bullet.endPosition[1] - bullet.startPosition[1]
norm = math.sqrt(direction[0] ** 2 + direction[1] ** 2)
if norm < 1e-4:
    direction = 0, 0
else:
    direction = direction[0] / norm, direction[1] / norm
```

Can you tell what it does? Can you tell if there is an error? If you are not convinced, think about a method of 200 lines with such computations!

The "Divide and conquer" rule is relevant at any code level, including implementing a method. If a method (or a function) becomes too long and too difficult to understand, it is the right moment to simplify it with new sub-methods (or sub-functions). In the code attached to this section, you can see other examples.

3.7.3.3 DeleteDestroyedCommand

The `DeleteDestroyedCommand` deletes all "dead" game items in a list:

```
1 class DeleteDestroyedCommand(Command):
2     def __init__(self, itemList):
3         self.itemList = itemList
4     def run(self):
5         newList = [item for item in self.itemList if item.alive]
6         self.itemList[:] = newList
```

We use this command with the `bullets` list of the `GameState` class; we could also use it with any other item list. About the implementation of this removal, we first create a new list of items where the `alive` item attribute is `True`:

```
newList = [item for item in self.itemList if item.alive]
```

This syntax called "list comprehension" is compact and equivalent to the following one:

```
newList = []
for item in self.itemList:
    if item.status:
        newList.append(item)
```

Line 6 also introduces a new syntax with the two dots in brackets `[:]`:

```
self.itemList[:] = newList
```

To update the `itemList` attribute, you could think about the following syntax:

```
self.itemList = newList
```

Using the second syntax, it updates the content of the `itemList` attribute. However, this attribute contains a *reference* to a list; it is not a list. So, without the `[:]`, the `itemList` attribute references a new list, and the previously referred list is not changed. With the `[:]`, we copy all items from the list referenced by `newList` into the list referenced by `self.itemList`. If these references are not clear, don't worry. It is a difficult topic for many new programmers (and actually, many programmers still don't understand them and sometimes don't even know that they exist...)

3.7.4 User Interface

The main change in the `UserInterface` is in the `processInput()` method, where we create the commands:

```
1 def processInput(self):
2     ...
3     if mouseClicked:
4         self.commands.append(ShootCommand(gameState, playerUnit))
5     for bullet in gameState.bullets:
6         self.commands.append(MoveBulletCommand(gameState, bullet))
7     self.commands.append(DeleteDestroyedCommand(gameState.bullets))
```

Lines 3-4 add a new shoot command when the player clicks the left mouse button (we set `mouseClicked` to `True` during the pygame event parsing at the beginning of the method). Remind that a command does not necessarily lead to an update. For instance, the shoot command does not create a bullet if the player is dead or shot recently.

Lines 5-6 add one move command for each bullet in the list: Note that we add commands in the order of the list. We could change this; for instance, we could move the player bullets before the others. We can see the benefits of this pattern: execution order and implementation are entirely separated. We can change one without worrying about the other.

Line 7 adds the command that removes all "dead" bullets in the `bullets` list of the game state This command is the last one to be executed (it is the last one in the list of commands). As a result, we take no risk, and no unexpected behavior can happen.

3.7.5 Other improvements

We also implement new features to handle bullets (see the attached code for details):

- `UnitsLayer` only draw units when their `alive` attribute is `True`;
- The new `BulletsLayer` class draws bullets; we add it to the list of layers in `UserInterface`;
- During input processing, `UserInterface` adds shoot commands when the player's unit is close to unplayed units;
- The game update in `UserInterface` increases the game state epoch by 1.

3.7.6 What to remember

Item life cycle: the Command pattern is an excellent way to handle objects' creation, update, and deletion. We create a command for each case and then schedule them effectively. For example, we run all the creation commands, the updates, and finally, the deletions.

Game epoch: it is the number of times we update the game state. We use it instead of time always to get the same gameplay, whatever the power of the player's computer.

Gather parameters: always store parameters (bullet speed, range, etc.) in variables or attributes, and put all of them in the same place.

Long method => refactor! when a method or a function becomes too long or too hard to read, create functions or methods to reduce it and write a more readable code.

List comprehension: we can filter the content of a list with the following syntax:

```
newList = [item for item in otherList if <condition>]
```

The new list will only contain items that satisfy the condition, for instance `item.attribute == value`.

In-place list update: We can update the content of a list with the following syntax:

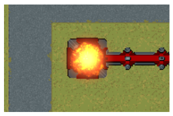
```
myList[:] = otherList
```

3.8 Animations

3.8.1 Functional and aesthetic animations

We added animated items connected to the game logic in the previous sections, like the tank or bullets. Then, we synchronized their visual state with their state in the game data. For instance, the location of a bullet in the game state is the same on the screen after converting cell coordinates to pixel coordinates. Thus, it leads to animated bullets that move in the world.

We can also add animations that we don't synchronize with the game state. For instance, when we destroy a unit, we can add an explosion:



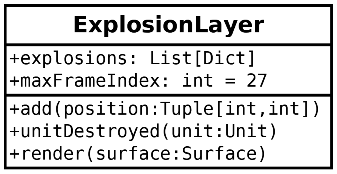
The game logic doesn't need this animation to work; it is purely aesthetic. If we want to respect our dividing of problems, these animations should not be in the game logic (state and commands). For the rendering part, we add a new layer for explosions. It contains a list of explosions and draws them as long as they exist. Then, we add an explosion every time we destroy a unit, for instance, using a `add()` method in the new layer.

A naive approach consists of calling this method directly from the game logic. However, by doing so, we introduce a circular dependency between state and rendering: the state depends on rendering classes and method names, and the rendering depends on state classes and method names. Expert programmers warn you: circular dependencies are a nightmare that leads to many problems. As for all the software design "golden rules", it is hard to prove with a few sentences. However, it is more straightforward for Python: we will have trouble with code calling methods defined after in the file!

To remove these dependencies, we use the Observer pattern. It is undoubtedly the most popular game pattern, right after the Game Loop pattern.

3.8.2 Explosions layer

The `ExplosionsLayer` class handles the life cycle of explosions:



3.8.2.1 Add an explosion

The `add()` method adds a new explosion to the `explosions` list:

```python
def add(self, position):
    self.explosions.append({
        'position': position,
        'frameIndex': 0
    })
```

We use a new syntax that adds a new item to the list that contains two sub-items, 'position' and 'frameIndex'. It is a nice feature of the Python language that allows us to create variables of any name at runtime. The 'position' key contains the location of the explosion on the screen and 'frameIndex' the current animation frame to display.

The curly brackets {} define a Python dictionary. It contains objects like lists, except that indexes (or keys) are not limited to integers. It can be more apparent if we split the creation of this dictionary from its addition to the list:

```python
explosion = {
    'position': position,
    'frameIndex': 0
}
self.explosions.append(explosion)
```

We could also split the creation and the filling of this dictionary:

```python
1 explosion = {}
2 explosion['position'] = position
3 explosion['frameIndex'] =  0
```

Line 1 is the creation of an empty dictionary. Lines 2-3 set values in this dictionary. We use brackets `[]` as with lists for getting or setting values. We will see more examples of dictionaries in the following chapters.

3.8.2.2 Render explosions

The `render()` method renders explosions, and delete finished ones:

```
1  def render(self, surface):
2      for explosion in self.explosions:
3          frameIndex = math.floor(explosion['frameIndex'])
4          position = explosion['position']
5          self.drawTile(surface, position, (frameIndex, 4))
6          explosion['frameIndex'] += 0.5
7
8      self.explosions = [
9          explosion for explosion in self.explosions
10         if explosion['frameIndex'] < self.maxFrameIndex
11     ]
```

Line 2 iterates through all explosions using a syntax we saw many times. Line 3 rounds the value of 'frameIndex'. We use float values to allow animations with a different rate than the screen.

Line 5 renders the tile. All frames are in the fourth row of the tileset, from the beginning to the end of the animation. If you want to propose different animations, you can create a new item in the explosion dictionary (like 'tilesRow') and update all methods accordingly.

Line 6 updates the current frame of the animation. It depends on the current frame rate and thus on the computer currently running the game. If powerful enough, it renders 60 frames per second, and animations are always the same. If it is not the case, animations last longer. We could get a more robust solution by introducing time computations (like elapsed time since the last frame). Note that the animation speed does not depend on game epochs! They live their own life, whatever happens to the game state (like a pause).

Lines 8-11 build a new list containing all unfinished explosions. The `ExplosionsLayer` is the only one to use this list; we can do this without risks. This "pythonish" code with list comprehension is equivalent to:

```
explosions = []
for explosion in self.explosions:
    if explosion['frameIndex'] < self.maxFrameIndex:
        explosions.append(explosion)
self.explosions = explosions
```

Remind that attributes and variables are different: `self.explosions` is an attribute, and `explosions` is a variable. They can refer (or not) to the same object. In this example, `explosions` first points to the new list, and `self.explosions` to the old one. Then, they both refer to the new list in the last line. Finally, since we no longer reference the old list, Python automatically destroys it.

3.8.3 Trigger explosions with the Observer pattern

If we define an explosion layer in the `UserInterface` class, adding an explosion starts the animation. For instance, if we add `self.layers[4].add((0,0))` after the creation of the layers list, an explosion appears at the top left corner of the screen.

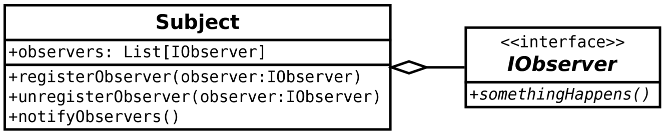
A naive way to trigger these animations is to call the `add()` method of the `ExplosionsLayer` class instance when a unit is destroyed (for example, in the `MoveBulletCommand` class). By doing so, the game state would depend on the layers. Since the layers already depend on the game state, we get a circular dependency. We don't like circular dependencies in software design and use them only if we have no choice.

The other issue with a game state that depends on layers is that it is hard to control the flow. For example, if the game state changes layers parameters during its updates, rendering should not happen since we could have strange results (and crashes in some cases). It is also challenging to handle situations where state and rendering updates are different. Finally, we can also add all the motivations for the Command pattern, which also gives better control of the flow.

We can remove the dependency between the game state and the layers with the Observer pattern. With this approach, the game state ignores the existence of the layers, and the layers updates themselves silently without disturbing the game state.

3.8.3.1 The Observer pattern

This pattern has two main actors, a class and an interface:


Subject
+observers: List[IObserver]
+registerObserver(observer:IObserver)
+unregisterObserver(observer:IObserver)
+notifyObservers()
<<interface>>
IObserver
+somethingHappens()

The `Subject` class is the observed one and holds a list of `observers`. Anyone can add or remove an observer in this list using the `registerObserver()` and `unregisterObserver()` methods. We call the `notifyObservers()` method when something happens: it notifies all observers on the list. Note that the `Subject` class and anyone outside this class can trigger the notification.

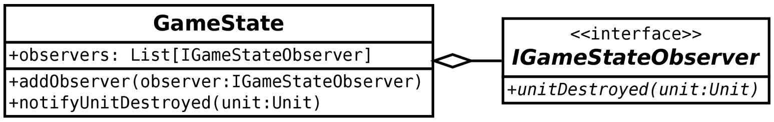
The `IObserver` class is an *interface* because it has no attributes and only methods. This interface aims to define all the events we can send to observers. The classes that implement `Observer` receive the notifications. For instance, we call the `somethingHappens()` method if something happens to the subject. There is a single notification in this example, but we can create as many cases as we want, with or without arguments, like `somethingCreated(index: int)` or `mouseMoved(info: MouseInfo)`. For each case, we need a corresponding notification method in the Subject class, like notifySomethingCreated(index: int) or notifyMouseMoved(info: MouseInfo).

The most exciting feature of this pattern is that the subject knows nothing about its observers. If something happens, someone triggers the notifications, and the subject's life goes on with no changes. There are no dependencies between the subject and the observer classes.

This pattern is typical in Graphic User Interface libraries, where controls (like push buttons) notify observers that something happens (like a button being pushed).

3.8.3.2 Observable game state

We use the Observer pattern to trigger the explosions (we only show related components):


GameState
+observers: List[IGameStateObserver]
+addObserver(observer:IGameStateObserver)
+notifyUnitDestroyed(unit:Unit)
<<interface>>
IGameStateObserver
+unitDestroyed(unit:Unit)

We create a new list attribute observers, add items to this list in addObserver() and iterate through all items in notifyUnitDestroyed():

```
class GameState:
    def __init__(self):
        ...
        self.observers = []
    def addObserver(self, observer):
        self.observers.append(observer)
    def notifyUnitDestroyed(self, unit):
        for observer in self.observers:
            observer.unitDestroyed(unit)
```

The IGameStateObserver interface contains a single method its children can implement:

```
class IGameStateObserver:
    def unitDestroyed(self, unit):
        pass
```

Note that the default behavior of the unitDestroyed() method is to do nothing: child classes that don't implement it don't react to this event.

3.8.3.3 Observe the game state

The Layer class inherits the IGameStateObserver interface, and its child classes can react when someone destroys a unit. The ExplosionsLayer class is the only interested one, so it is the only one to implement the `unitDestroyed()` method. It adds an explosion that starts at the unit location:

```
class ExplosionsLayer(Layer):
    ...
    def unitDestroyed(self, unit):
        self.add(unit.position)
```

Finally, something needs to trigger the notification. The best one for that is the one who destroys it: the `MoveBulletCommand` class. More specifically, we add a single line `state.notifyUnitDestroyed(unit)` in the `run()` method when we destroy the unit:

```
class MoveBulletCommand(Command):
    ...
    def run(self):
        ...
        unit = state.findLiveUnit(newCenterPos)
        if unit is not None and unit != bullet.unit:
            bullet.alive = False
            unit.alive = False
            state.notifyUnitDestroyed(unit)
            return
```

And that's all! Thanks to the call to `notifyUnitDestroyed()`, the explosion layer gets notified and creates a new explosion animation. Since the explosions layer works independently, there is no more to do.

3.8.4 Optimizations

The current frame rate of the game may be slow on most computers because Pygame is not well optimized. It is an excellent library for beginners but not the best one for optimal user experience. Any call to a Pygame function is usually costly: we can speed up the rendering if we find a way to reduce the number of these calls. Furthermore, blitting surfaces with an alpha channel is slow: we should avoid it as much as possible.

We can reduce the number of Pygame function calls by saving the rendering of background layers. The background never changes; we don't need to redraw it every frame. Furthermore, the first background doesn't need an alpha channel; we can create a surface with no transparency for this case. We update the `ArrayLayer` class with these optimizations.

We first add two new attributes to `ArrayLayer`:

- `surface`: a Pygame surface that contains the rendered layer. If it is `None`, it means we did not render yet, or we should refresh the rendering.

- `surfaceFlags`: flags for the creation of a Pygame surface. We are interested in two cases: no alpha channel (0) and with alpha channel (`pygame.SRCALPHA`).

We also update the `render()` method:

```python
1  def render(self, surface):
2      if self.surface is None:
3          self.surface = pygame.Surface(
4              surface.get_size(), flags=self.surfaceFlags
5          )
6          for y in range(self.state.worldSize[1]):
7              for x in range(self.state.worldSize[0]):
8                  tile = self.array[y][x]
9                  if tile is not None:
10                     self.drawTile(self.surface, (x, y), tile)
11     surface.blit(self.surface, (0, 0))
```

If the `surface` attribute is `None` (line 1), we create a new surface (lines 3-5) and render the layer into it (lines 6-10). Note that `self.surface` is the class attribute, and `surface` is the method argument! The call at line 11 is as before, except that we replace `surface` with `self.surface`. It blits the `surface` attribute into the `surface` argument: this is a single Pygame call, faster than hundreds of them.

Finally, when we create the layers in the constructor of the `UserInterface` class, all array layers automatically benefit from the caching of their rendering. Furthermore, the first layer has a surfaceFlags=0 value to create a surface with no alpha channel, which speeds up the rendering. We should get 60 frames per second on most computers thanks to these optimizations.

3.8.5 What to remember

Game logic and rendering: they both can lead to changes on the screen, like moving items. However, they live in two different time-spaces: the former lives with game data and epochs and the latter with screen pixels and frame rates. Furthermore, the game logic should live without the rendering: we can run it on a server with no screen. On the other side, the rendering illustrates the current state of the game data. Thus, it can reflect exactly this data (ex: bullets) or add more content (ex: explosions).

Observer pattern: it allows an *observable* object to notify any registered *observer* about events. The main purpose of this pattern is to connect objects without introducing a circular dependency. Game logic and rendering is good example. The rendering depends on game logic since its attributes and methods are used to get the game data. However, game logic can not do the same. Here comes the pattern: when the game logic does something of interest, it notifies its observers. If the rendering is an observer, it receives the notification and can act accordingly.

Python dictionaries: we create them with curly brackets `{}` and access items using brackets `[]`:

```
data = {
    "name": "Peter",
    "age": 23
}
print(data["name"], data["age"])
```

CHAPTER 4

Around the game

In the previous chapter, we created a small tank game. It is a complete game: we can move the tank, fire, destroy enemies, or get destroyed. However, nothing happens when we kill all towers or lose, and we can not restart or choose another level.

When creating games and applications, we also need to develop extra features that are not the game itself. Although fully necessary, one can neglect these parts because they do not look useful. This chapter shows examples of these “game extras” like levels and menus. Then, as before, we introduce new Python features and software design techniques to implement them.

You can find examples from this chapter here: `https://github.com/philippehenri-gosselin/discoverpythonpatterns/tree/master/chap04`. The names of the examples begin with the corresponding section. All the tileset images are in this folder.

To run the example in this chapter, we use a free library called *tmx*. You can install it using `pip`: open an Anaconda prompt (as we did for Pygame), and run:

```
pip install tmx
```

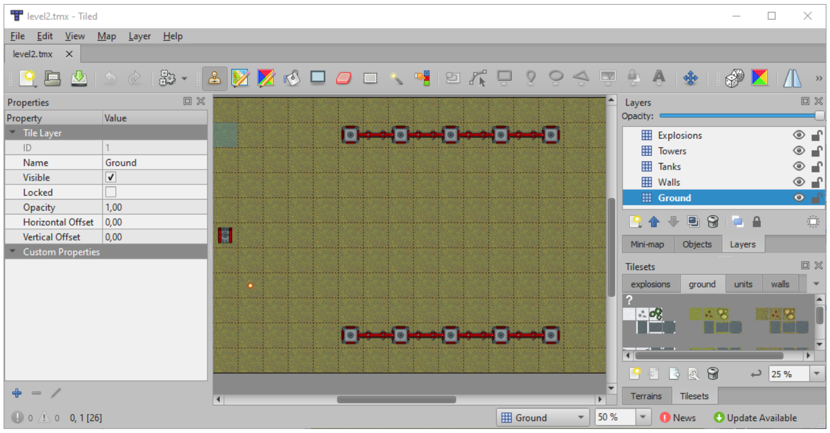
At the beginning of our program, add the following line to import it:

```
import tmx
```

4.1 Create and load a level

4.1.1 Create a new level with Tiled Map Editor

We create a level with *Tiled Map Editor*. It is free software you can download here: `https://thorbjorn.itch.io/tiled`.



level2.tmx - Tiled

Property	Value
Tile Layer	
ID	1
Name	Ground
Visible	✓
Locked	
Opacity	1,00
Horizontal Offset	0,00
Vertical Offset	0,00
Custom Properties	

It allows for creating maps based on tiles; Its interface is the following one:

- The top bar below the menu shows all the current maps. In this screenshot, there is a single map “level2.tmx”;
- To the left, we can see the properties of an item. In this screenshot, we see the properties of the ground layer;
- In the center, we can see the map;
- Top right, there is a list of layers. In this screenshot, we can count five layers;
- Bottom right, there are all the available tilesets.

Create a new level: in the menu, **select File / New / New Map...**.

In the “Map” section, **select Orthogonal** orientation.

In the “Map size” section, **select Fixed** and choose the width and height of your level (number of tiles per row and column, not pixel!)

In the “Tile size” box, **type 64 px** for width and height (the pixel size of our tiles).

Click Save as..., and **select** the location and name of your level (.tmx file).

4.1.2 Layers and tilesets

There is a list of layers at the top right of the tiled interface (If it is not the case, select the layers tab). We will add the following one:

- Ground layer: decorative items we can walk on
- Walls layer: impassable decorative items

- Tanks layer: location of tanks
- Towers layer: location of towers
- Explosions layer: top animations

Ground and walls layers correspond to the two first layers in our game (items of index 0 and 1 in the `layers` list of `UserInterface`).

Tanks and towers layers contain the locations of all units and correspond to the third layer in our game (the item of index 2 in the `layers` list). We use two layers in Tiled to know who is playable and who is an enemy.

The explosions layer contains the top animations. We use it to define the tileset for the bullets and the explosions and correspond to the two last layers in our game.

Create the layers: let's create these layers in Tiled. First, rename the first layer to "Ground": **double click** on the current name ("Tile layer 1") and **type "Ground"**. Note that naming the layers is not mandatory, but it helps during the edition of the map.

Add a new layer using the first icon in the layers toolbar (below the list of layers). A menu appears: **select Tile Layer**. **Rename** this second layer to "Walls".

Note that this second layer appears on top of the first. It means that, during rendering, Tiled first draws the bottom layer ("Ground"), and then the top layer ("Walls"), as in our game. If you don't set the proper order, one layer can hide all the others.

Repeat for all the remaining layers: "Tanks", "Towers" and "Explosions".

Define the tilesets: bottom right, **click** the first icon in the tilesets toolbar (below the tilesets). A dialog appears.

Click Browse... and **select** the "gound.png" image file containing the background tileset. This file, as well as the others, is available at `https://github.com/philippehenri-gosselin/discoverpythonpatterns/tree/master/chap04`.

Make sure that "Embed in map" is checked, and **click OK**.

Repeat for "walls.png", "units.png" and "explosions.png".

4.1.3 Edit the level

You can edit the level in the center part of the Tiled interface.

First, **select** the layer you want to work on by left-clicking on its name. Then, **select** one of the tilesets in the bottom right part.

Finally, **choose** one tile and **start drawing** the map. If you can't draw, ensure that you selected the "Stamp Brush" in the top bar. If you need to remove a tile, choose the "Eraser" in the top bar.

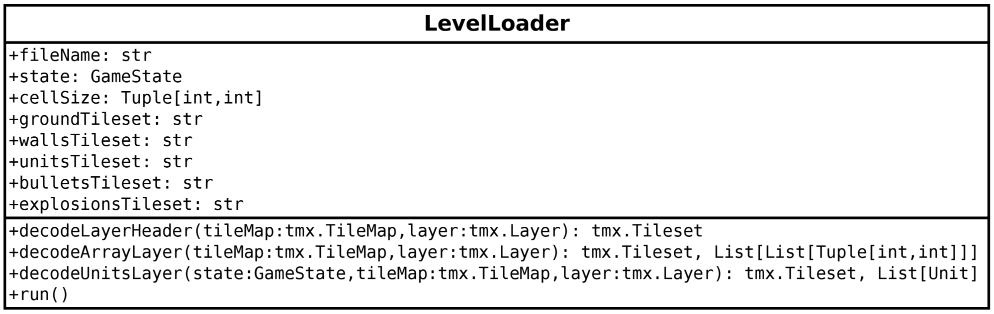
Note that Tiled allows you to use tiles from any tileset for any layer. It is not the case for our game, where a layer can only use tiles from a single tileset. When creating the level, please pay attention to this; otherwise, we won't be able to load it.

You can now create the level. For the tanks layer, put a single tank tile to define the player's starting position. We could put several tanks, but our program can't handle several players or several tanks controlled by one player for now (it is not difficult to implement thanks to our design, you can try it as an exercise).

For the explosions layer, put a single tile from the explosions tileset. It defines which tileset is used by the explosions layer in our program.

4.1.4 Level loader

LevelLoader class: we create a new `LevelLoader` class that parses a ".tmx" file and stores the resulting data in its attributes:



```
LevelLoader
+fileName: str
+state: GameState
+cellSize: Tuple[int,int]
+groundTileset: str
+wallsTileset: str
+unitsTileset: str
+bulletsTileset: str
+explosionsTileset: str
+decodeLayerHeader(tileMap:tmx.TileMap,layer:tmx.Layer): tmx.Tileset
+decodeArrayLayer(tileMap:tmx.TileMap,layer:tmx.Layer): tmx.Tileset, List[List[Tuple[int,int]]]
+decodeUnitsLayer(state:GameState,tileMap:tmx.TileMap,layer:tmx.Layer): tmx.Tileset, List[Unit]
+run()
```

Attributes are:

- `fileName` contains the name of the file to read;
- `state` refers the game state created by the loader;
- `tileSize` is a tuple with the width and height of tiles;
- `xxxxTileset` contains the name of the tileset for layer `xxxx`.

Methods are:

- `decodeLayerHeader()`: checks the properties of a tmx layer; returns a tileset's name;
- `decodeArrayLayer()`: returns a 2D array of tuples with tile coordinates and a tileset's name;
- `decodeUnitsLayer()`: returns a list of units and a tileset's name;
- `run()`: parses all the ".tmx" content and fills the attributes with data.

The run() method: to load a new level, we create an instance of this class and call the `run()` method'. This method begins with the loading of the ".tmx" file:

```
1 def run(self):
2     if not os.path.exists(self.fileName):
3         raise RuntimeError("No file {}".format(self.fileName))
4     tileMap = tmx.TileMap.load(self.fileName)
5     ...
```

Lines 2 checks that the file exists. If it is not the case, we raise an exception (line 3), telling that the file does not exist. The `exists()` function of the `os.path` standard library returns `True` if the file exists, `False` otherwise. Don't forget to import `os` to get access to this function.

In the exception message, we can recognize the string formatting syntax, where the braces `{}` are replaced by the content of the argument in the following `format()` method. We can see the method call of `"No file{}"`, which is an instance of the Python string class. It is easier to understand if we introduce intermediate variables:

```
formatString = "No file {}"
message = formatString.format(self.fileName)
raise RuntimeError(message)
```

Line 3 calls the `load()` static method of the `tmx.TileMap` class:

```
tileMap = tmx.TileMap.load(self.fileName)
```

We can call static methods without an instance, like functions. Note that we need to use the name of the class in place of the instance. In this example, the class is `tmx.TileMap`. If it were `MyClass`, a call to a static method `myMethod()` would be `MyClass.myMethod()`.

The returned value is an instance of the `tmx.TileMap` class. You can find a detailed description of this class here: `http://python-tmx.nongnu.org/doc/`.

Check, check, check! Loading an external file is always risky. Even if we are loading a file we created with the software we own, there is still a risk. For instance, if our software updates itself and introduces changes in the file format, we could face unexpected issues. It is even more problematic when other team members create these files. And finally, if we allow players to load their levels, many cases can happen.

It is the reason why we should add as many checks as we can, including checks that seem obvious. If something wrong happens, it will warn the player that the file format is not supported. In some cases, the cause can be understandable for the player, and (s)he will be able to correct the level. For instance, if the format expects five layers, and the player creates a level with six layers, (s)he can fix it. All other warnings will help the developer team better than a crash or unexpected behavior.

Most of the content of the `LevelLoader` class is similar: it reads a value, checks if it is correct, and raises an exception in case of error. In the following, we only detail what presents interesting programming concepts.

4.1.5 Split long methods

We could parse all the data in the ".tmx" file with a single long method. As before, we split these long code blocks into several smaller methods. In this case, we put the decoding of array-based layers in the `decodeArrayLayer()` method and the decoding of unit-based layers in `decodeUnitsLayer()`.

Decode array-based layers: we create the ground layer in the following way:

```
tileset, array = self.decodeArrayLayer(tileMap, tileMap.layers[0])
self.tileSize = (tileset.tilewidth, tileset.tileheight)
state.ground[:] = array
self.groundTileset = tileset.image.source
```

The `decodeArrayLayer()` returns the corresponding `tmx.Tileset` and the array of tile coordinates (as used in the game state).

The following lines work the same, except that we decode the second layer:

```
tileset, array = self.decodeArrayLayer(tileMap, tileMap.layers[1])
if tileset.tilewidth != tileSize[0] \
or tileset.tileheight != tileSize[0]:
    raise RuntimeError("Error in {}: ...".format(self.fileName))
state.walls[:] = array
self.wallsTileset = tileset.image.source
```

We use the `decodeArrayLayer()` method twice: it reduces the number of lines and avoids the need to maintain two similar code blocks.

Decode layer headers: when we coded the `decodeArrayLayer()` and `decodeUnitsLayer()` methods, the beginning was the same. Consequently, we created a new method with this common code. For instance, in both cases, we have to check that the layer object is an instance of the `Layer` class from the `tmx` package:

```
if not isinstance(layer, tmx.Layer):
    raise RuntimeError("Error in {}: ...".format(self.fileName))
    ...
```

The built-in `isinstance()`function checks that the object (first argument) inherits a class (second argument). It means that it can be the class or any child class. Thus, it is different from the expression `type(layer) == tmx.Layer`, which returns `True` if and only if the object is an instance of the class.

We also check that the number of tiles in the layer is as expected:

```python
if len(layer.tiles) != tileMap.width * tileMap.height:
    raise RuntimeError("Error in {}: ...".format(self.fileName))
```

The built-in `len()`function returns the number of items in a container (list, dictionary, etc.).

4.1.6 2D array and 1D/2D conversion

The `decodeArrayLayer()` method decodes a `tmx.Layer` instance, and returns the corresponding tileset and the 2D array of tile coordinates:

```python
1  def decodeArrayLayer(self, tileMap, layer):
2      tileset = self.decodeLayerHeader(tileMap, layer)
3  
4      array = [None] * tileMap.height
5      for y in range(tileMap.height):
6          array[y] = [None] * tileMap.width
7          for x in range(tileMap.width):
8              tile = layer.tiles[x + y * tileMap.width]
9              if tile.gid == 0:
10                 continue
11             lid = tile.gid - tileset.firstgid
12             if lid < 0 or lid >= tileset.tilecount:
13                 raise RuntimeError("...".format(self.fileName))
14             tileX = lid % tileset.columns
15             tileY = lid // tileset.columns
16             array[y][x] = (tileX, tileY)
17 
18     return tileset, array
```

Create a 2D array: the method creates a 2D array in the `array` variable using a list of lists. It is a typical process in Python (and programming in general). We first create a list with `None` values (line 4). Then, we iterate between 0 and the array's height minus one (line 5). In the loop, we create a list of `None` values for each layer row (line 6). Consequently, we create a total of (height + 1) lists. Finally, to fill the 2D array with values (other than `None`), we run a second loop between 0 and the width of the array minus one (line 7) and use the syntax `array[y][x] = value`, as in line 16.

1D/2D conversion: there are other typical calculations in this method:

- Convert 2D coordinates (x,y) into 1D: `x + y*width`. In this example, the width is the one of the map: `tileMap.width` (line 8);
- Convert 1D coordinates `index` into 2D: `x = index % width` and `y = index // width` (`//` is the integer division). In this example, we convert the 1D tile identifier (`lid`) into 2D tile coordinates `tileX` and `tileY` (lines 14-15).

4.1.7 What to remember

Check data from files: when we read data from a file, we should do as many checks as possible.

Check object class: the `isinstance(object, class)` function returns `True` if `object` is an instance of `class` or one of its children.

Get the length of a container: the `len(container)` function returns the number of items in a container.

Create a 2D array: we can use the following algorithm to create and fill a 2D array made with lists:

```
array = [None] * height
for y in range(height):
    array[y] = [None] * width
    for x in range(width):
        ...compute value...
        array[y][x] = value
```

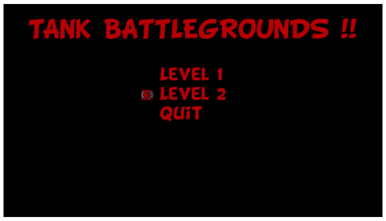
Convert 1D to 2D: the expression is `index = x + y*width`, where `x,y` are the 2D coordinates, and `index` the corresponding 1D coordinate.

Convert 2D to 1D: the expressions are:

```
x = index % width
y = index // width
```

where `index` is the 1D coordinate, and `x,y` the corresponding 2D coordinates.

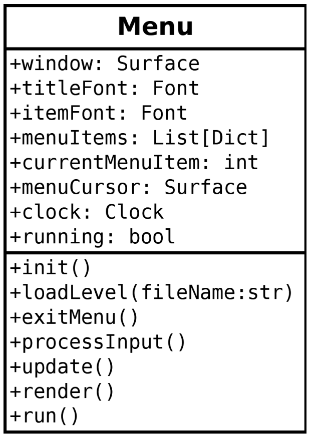
4.2 Game menu



This section creates a menu with Pygame and the Game Loop pattern. We create a new program (independent from the previous one); we will add the menu to the tank game in the next section.

4.2.1 Menu with the Game Loop pattern

We create the menu following the Game Loop pattern. This approach allows us to use it later inside our game. We create the following class that embeds all that we need for running the menu:



Attributes:

- `window`: a Pygame window;
- `titleFont` and `itemFont`: Pygame fonts for the title and menu items;
- `menuItems`: the content of the menu (more details below);
- `currentMenuItem`: the index of the currently selected menu item;
- `menuCursor`: a Pygame surface for the menu cursor (a tank pointing to the currently selected item);
- `clock`: a Pygame clock to limit the frame rate (60Hz);
- `running`: `True` until the end of the program.

Methods:

- `init()`: the initialization method called once. We create the window, the fonts, the menu items, and the variables that describe the current state of the menu;
- `loadLevel()`: we call it when the player selects a level in the menu. For now, it only displays a message in the console;
- `exitMenu()`: we call it when the player selects the `Quit` menu item. It sets the `running` attribute to `False`;
- `processInput()`: parses Pygame events to find key presses and act accordingly. Up/Down arrow keys increase/decrease `currentMenuItem` and the Return key triggers the action corresponding to the currently selected item;
- `update()`: does nothing because the menu is so simple that we don't need to follow the Command pattern. It is better to create it anyway, in case our game evolves and we need more advanced features;
- `render()`: renders the menu;
- `run()`: the main game loop.

4.2.2 Represent menu items

We store all the menu item data using several exciting features of the Python language:

```
self.menuItems = [
    {
        'title': 'Level 1',
        'action': lambda: self.loadLevel("level1.tmx")
    },
    {
        'title': 'Level 2',
        'action': lambda: self.loadLevel("level2.tmx")
    },
    {
        'title': 'Quit',
        'action': lambda: self.exitMenu()
    }
]
```

First, the `menuItems` attribute is a list because we use brackets `[]`. A comma separates each item of this list:

```
self.menuItems = [ item1, item2, ... ]
```

Each item of the list is a dictionary because of the curly brackets `{}`. For instance, the first item is:

```
{
    'title': 'Level 1',
    'action': lambda: self.loadLevel("level1.tmx")
}
```

Each element of the dictionary has a unique key. In this example, the first key is ‘title’ and the second is ‘action’. We could update the content of this first menu item in the following way:

```
self.menuItems[0]['title'] = 'Level 1'
self.menuItems[0]['action'] = lambda: self.loadLevel("level1.tmx")
```

We can stack indices: `[0]` is the index of the first item in the `menuItems` attribute, and `['title']` is the index of the “title” item in the `self.menuItems[0]` dictionary.

About values, the one of ‘title’ is the message to display for the menu item. The content of ‘action’ is new: it is a lambda function. It is the function to call when the player asks for its execution.

Lambda functions are powerful features available in most programming languages. They allow the creation of anonymous inline functions. Without lambdas, we must create a function and find a unique name:

```
def loadLevel1():
    menu.loadLevel("level1.tmx")
self.menuItems[0]['action'] = loadLevel1
```

We save several code lines and get a more readable syntax. Moreover, it can transparently embed variables, in which case the lambda is a *closure*. In this example, the function automatically embeds the `self` variable. Thus, when we call the lambda, it knows who is `self`, and it correctly refers to the menu instance.

4.2.3 The processInput() method

This method parses the Pygame events list to move the cursor and trigger menu actions:

```
1  def processInput(self):
2      for event in pygame.event.get():
3          if event.type == pygame.QUIT:
4              self.exitMenu()
5              break
6          elif event.type == pygame.KEYDOWN:
7              if event.key == pygame.K_DOWN:
8                  if self.currentMenuItem < len(self.menuItems) - 1:
9                      self.currentMenuItem += 1
10             elif event.key == pygame.K_UP:
11                 if self.currentMenuItem > 0:
12                     self.currentMenuItem -= 1
13             elif event.key == pygame.K_RETURN:
14                 menuItem = self.menuItems[self.currentMenuItem]
15                 try:
16                     menuItem['action']()
17                 except Exception as ex:
18                     print(ex)
```

Lines 3-5 call the `exitMenu()` method if the `QUIT` event is triggered (for instance, if the user clicks the window's close button).

Lines 7-12 handle arrows keys and update the `currentMenuItem` attribute accordingly.

Lines 14-18 are the most interesting. If the player presses the return key, the `menuItem` refers to the current menu item (line 14). Line 16 calls the function stored in the `action` key. We could write it with an intermediate variable:

```
actionFunction = menuItem['action']
actionFunction()
```

Note that actionFunction as well as menuItem['action'] don't contain the function. As for objects, they *refer* to the function. These lines never copy any function, only references.

A try...except block surrounds line 16. Thanks to this syntax, Python executes the block following the except statement if an error occurs. In this example, we print the exception's message (line 18) to help us find the bug. Note that the error can appear anywhere inside the block. It includes inside a function, a function called by a function, etc. In all cases, execution stops and goes straight to the block following the except statement.

4.2.4 The render() method

This method renders the menu and contains several interesting parts.

Draw a centered text: the following code draws the title at height y and centered horizontally:

```
1 surface = self.titleFont.render(
2    "TANK BATTLEGROUNDS !!", True, (200, 0, 0)
3 )
4 x = (self.window.get_width() - surface.get_width()) // 2
5 self.window.blit(surface, (x, y))
```

Lines 1-3 create a new Pygame surface with the text. It uses the render() method of the Font class. The first argument in the text to render, the second enables antialias, and the last is the color.

Line 4 computes the x coordinates to center the text. We compare the width of the window self.window.get_width() with the width of the surface with the text surface.get_width(). Finally, we divide the difference by 2.

Line 5 draws the surface in the window.

Draw a centered menu: it is a bit more complicated because we have several text lines. The trick is to compute the largest width of those lines:

```
1     menuWidth = 0
2     for item in self.menuItems:
3         surface = self.itemFont.render(item['title'], True, (200, 0, 0))
4         menuWidth = max(menuWidth, surface.get_width())
5     x = (self.window.get_width() - menuWidth) // 2
```

This code is a typical implementation of a maximum computation: we init a variable for the maximum value (line 1), iterate through all items (line 2), compute the value of one item (line 3) and update the maximum (line 4).

We can get the same result with a list comprehension:

```
menuWidth = max([
    self.itemFont.render(item['title'], True, (200, 0, 0)).get_width()
    for item in self.menuItems
])
```

Don’t worry if you don’t understand this code; it uses advanced features.

4.2.5 What to remember

Element properties: we can use dictionaries to represent different application elements properties (like the menu items). We already saw a case with the explosions: we used a dictionary to store the properties (position and frame index). We could create a class to hold these properties, but it leads to more code for the same result and less readability. The first drawback of this approach is a (possible) overhead: it works fine when we have a few elements. It is also less attractive when we have a lot of properties or if types of properties are complex.

Draw text with Pygame: we draw text with the render method of the `Font` class:

```
font = pygame.font.Font("font.ttf", 24)
surface = font.render("message", True, (255, 255, 255))
```

We should create the font only once. The `Font` constructor expects the name of the file with the font (ex: “font.ttf”) and the size (ex: 24).

Instances of the `Font` class create a surface with the `render()` method. Its arguments are the text to render, `True` if we want antialias (smooth edges), and a color (red, green, blue).

Lambda functions: we create anonymous functions with the `lambda` keyword:

```
f = lambda: print("ok")
f()  # print "ok"
```

Lambda functions can have arguments and return a value:

```
f = lambda x: x + 1
print(f(3))  # print 4
```

Compute the maximum value in a container: the `max()` function returns the largest value in a container (list, dictionary, etc.).

Compute the minimum value in a container: the `min()` function returns the smallest value in a container (list, dictionary, etc.).

Advanced feature: Closures: functions keep references to variables declared in their environment:

```python
def f(x):
    def g(y):
        return x + y
    return g
h = f(3)
print(h(5))  # print 8
```

Advanced feature: Computation with list comprehensions: we can do a calculation with a list comprehension:

```python
result = max([2 * x for x in range(10)])
print(result)  # print 18
```

This syntax can run faster than a `for` loop in some cases.

4.3 Game modes

We introduce game modes to merge the menu in the previous section with the tank game. We can see them as an extension of the Game Loop pattern: each mode implements the pattern methods.

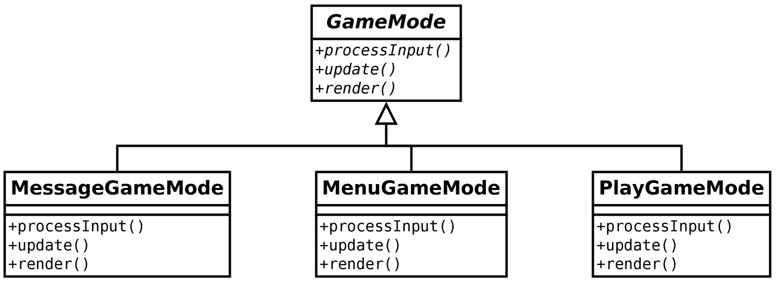
We create the three followings game modes:

- The default game, where the player controls a tank;
- A menu, where the player selects an option, like load a level or quit the game;
- A message.

Furthermore, we want to mix these modes. So when the menu or the message mode pops up, the play game mode should still be visible and in a frozen state.

4.3.1 A mode is a game

We create these game modes as if we were creating three games that use the Game Loop pattern (we don't show the attributes):



The three child class inherits the `GameLoop` base class with the methods of the Game Loop pattern: `processInput()`, `update()` and `render()`. It means we can run each mode with an instance of the corresponding child class without knowing its implementation. For instance, we can run one of them using a code that looks like this one:

```
running = True
clock = pygame.time.Clock()
gameMode = MenuGameMode(...) # or PlayGameMode(...)
                             # or MessageGameMode(...)
while running:
    gameMode.processInput()
    gameMode.update()
    gameMode.render()
    pygame.display.update()
    clock.tick(60)
```

4.3.2 Overlay

Thanks to this approach, we can also mix the different modes as if we were running several parallel games.

UserInterface class: we implement this feature in the `UserInterface` class, where we introduce new attributes:

- `currentActiveMode`: a string with the current mode: “Play” or “Overlay”;
- `overlayGameMode`: the current game mode to overlay (if not `None`). It can be an instance of `MenuGameMode` or `MessageGameMode`;
- `playGameMode`: the current game mode to play (if not `None`).

Then, the `run()` contains the main game loop with three parts:

```python
def run(self):
    while self.running:
        # Process input and update state...
        # Render in a surface...
        # Rescale and blit to screen...
```

Process input and update state: the first part of the loop in the run() method switches the control between the game and the overlay:

```python
1  if self.currentActiveMode == 'Overlay':
2      self.overlayGameMode.processInput()
3      self.overlayGameMode.update()
4  elif self.playGameMode is not None:
5      self.playGameMode.processInput()
6      try:
7          self.playGameMode.update()
8      except Exception as ex:
9          print(ex)
10         self.playGameMode = None
11         self.showMessage("Error during the game update...")
```

If the currently active mode is an overlay one (line 1), we call the `processInput()` and `update()` methods of this game mode (lines 2-3). In the case of the menu game mode, the player can control the cursor and select an image. In the case of the message game mode, the player can hit space or enter to continue.

If the currently active mode is the game (line 4), we use the `PlayGameMode` class (lines 5-11). On the contrary to the previous case, we expect errors to happen. Therefore, it is better to display an error message to the player rather than crashing the application. We could also implement more robust behaviors, like restoring the last saved game.

The `try ... except` statement surrounds the lines to watch. In this example, we catch exceptions raised during the game state update. If this is the case, the code flow goes directly to the `except` block.

The `except` block displays the exception's message to help developers figure out why something is wrong (line 9). Then, it deletes the current play game mode (line 10) and displays a message (line 11). The `showMessage()` enables a message mode overlay with the text in the argument.

Rendering: the second part of the `run()` method can render two "games" at the same time:

```
1  renderSurface = Surface((self.renderWidth, self.renderHeight))
2  if self.playGameMode is not None:
3      self.playGameMode.render(renderSurface)
4  else:
5      renderSurface.fill((0, 0, 0))
6  if self.currentActiveMode == 'Overlay':
7      darkSurface = pygame.Surface((renderWidth, renderHeight),
8                                   flags=pygame.SRCALPHA)
9      darkSurface.fill((0, 0, 0, 150))
10     renderSurface.blit(darkSurface, (0, 0))
11     self.overlayGameMode.render(renderSurface)
```

We first render the game if there is one or fill the screen with black if there is none (lines 2-5).

If the currently active mode is an overlay (line 6), we darken the background (lines 7-10) and render the overlay (line 11).

We create the darkening with a surface with an alpha channel (lines 7-8) and fill it with black and transparency (line 9). The color (0,0,0,150) means red=0, green=0, blue=0 and alpha=150. An alpha of 0 is fully transparent, and 255 is fully opaque.

4.3.3 Play game mode

The `PlayGameMode` class runs the game as before. It is like the previous `UserInterface` class, except that we remove the main game loop and handle the end of the game.

In the `update()` method, we handle the end of the game, right after the game update:

```
if not self.playerUnit.alive:
    self.gameOver = True
    self.ui.showMessage("GAME OVER")
else:
    oneEnemyStillLives = False
    for unit in self.gameState.units:
        if unit == self.playerUnit:
```

```
8               continue
9           if unit.alive:
10              oneEnemyStillLives = True
11              break
12      if not oneEnemyStillLives:
13          self.gameOver = True
14          self.ui.showMessage("Victory !")
```

If the player is dead (line 1), the `gameOver` attribute is set to `True` (line 2). We use this attribute in the `processInput()` method to ignore player commands. We also display a “GAME OVER” message (line 3). It uses the `showMessage()` method of the `UserInterface` class that activates a message game mode with the message in the argument.

If the player is still alive (line 4), we iterate through all enemies to see if there is still one alive (lines 5-11). Finally, if all enemies are dead (line 12), the game is over, and we display a victory message (lines 13-14).

4.3.4 Menu game mode

The `MenuGameMode` class is similar to what we created in the previous section to implement a menu. The main difference is the list of items and what they trigger:

```
self.menuItems = [
    {
        'title': 'Level 1',
        'action': lambda: self.ui.loadLevel("level1.tmx")
    },
    {
        'title': 'Level 2',
        'action': lambda: self.ui.loadLevel("level2.tmx")
    },
    {
        'title': 'Level 3',
        'action': lambda: self.ui.loadLevel("level3.tmx")
    },
    {
        'title': 'Quit',
        'action': lambda: self.ui.quitGame()
    }
]
```

Each menu item calls a method of the `UserInterface` when selected. For instance, to load a level, we call the `loadLevel()` method of the `UserInterface`:

```
def loadLevel(self, fileName):
    if self.playGameMode is None:
        self.playGameMode = PlayGameMode(self)
    try:
        self.playGameMode.loadLevel(fileName)
        self.renderWidth = self.playGameMode.renderWidth
        self.renderHeight = self.playGameMode.renderHeight
        self.playGameMode.update()
        self.currentActiveMode = 'Play'
    except Exception as ex:
        print("Error:", ex)
        self.playGameMode = None
        self.showMessage("Level loading failed :-(")
```

We call a `UserInterface` method rather than doing the job in the `MenuGameMode` class. The required changes are beyond the scope of the `MenuGameMode` class: its role is to manage a menu and not handle the game modes.

4.3.5 Message game mode

The `MessageGameMode` class is the most basic “game”. It only displays a message and waits for some keys to be pressed:

```
class MessageGameMode(GameMode):
    def __init__(self, ui, message):
        self.ui = ui
        self.font = pygame.font.Font("BD_Cartoon_Shout.ttf", 36)
        self.message = message
    def processInput(self):
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                self.ui.quitGame()
                break
            elif event.type == pygame.KEYDOWN:
                if event.key == pygame.K_ESCAPE \
                        or event.key == pygame.K_SPACE \
                        or event.key == pygame.K_RETURN:
                    self.ui.showMenu()
    def update(self):
        pass
    def render(self, surface):
        textSurface = self.font.render(self.message, True, (200,0,0))
        x = (surface.get_width() - textSurface.get_width()) // 2
        y = (surface.get_height() - textSurface.get_height()) // 2
        surface.blit(textSurface, (x, y))
```

4.3.6 What to remember

Game modes management: a first approach to handle game modes (like menus or different game UIs) is to add a new attribute with the mode's name. Then, depending on this attribute, we run the correct procedure in the Game Loop pattern methods.

When a mode is complex, it is better to create a new class that implements the methods of the Game Loop pattern. Then, a main class with the main game loop calls these methods.

Errors should not crash the game: errors can appear, even in a well-tested application. As much as possible, we should try to avoid game crashes and offer the player something to do, like reload the game.

Don't mix features in classes: each class should implement a specific feature. In our example, the `UserInterface` class manages game modes' life cycle but does not implement these modes. The `PlayGameMode` runs the game and should not create other modes (but can ask for their creation).

Draw a transparent rectangle: create a new Pygame surface with an alpha channel, fill it with a color with an alpha value, and blit this surface.

4.4 Music and sounds

This section adds music and sounds to our tank game and uses the Observer pattern to implement it efficiently.

We want to get the following result:

- Play music when the game starts;
- Play music when a level is loaded;
- Play music when the player wins;
- Play music when the player loses;
- Play a sound when a unit fires;
- Play a sound when a unit is destroyed.

The music is from `https://www.freesfx.co.uk`, and is all royalty-free (`https://www.freesfx.co.uk/Music.aspx`). We keep the original file names to better credit the authors:

- “17718_1462204250.mp3”: Menu music, called “Gang War”;
- “17687_1462199612.mp3”: Level music, called “Military Madness”;
- “17382_1461858477.mp3”: Victory music, called “Opening Day”;
- “17675_1462199580.mp3”: Game over music, called “Deadly Talk”.

The sounds come from `https://freesound.org`, and their license is Attribution-NonCommercial 3.0 Unported (`https://creativecommons.org/licenses/by-nc/3.0/`). We also keep the original file names:

- "170274__knova__rifle-fire-synthetic.wav": Bullet fire sound, created by knova (`https://freesound.org/people/knova/`);
- "110115__ryansnook__small-explosion.wav": Explosion sound, created by ryansnook (`https://freesound.org/people/ryansnook/`).

4.4.1 Music and sound with Pygame

Music: it is easy to play music with Pygame. First, load the music file using the `pygame.mixer.music.load()` function:

```
pygame.mixer.music.load("music.mp3")
```

Note that it stops any music already playing. Also, note that we can only have one piece at a time.

Once the music file is loaded, we can start with the `pygame.mixer.music.play()` function. This function has several optional arguments. In our case, we set `loops` to -1 to repeat indefinitely:

```
pygame.mixer.music.play(loops=-1)
```

You can find more details here: `https://www.pygame.org/docs/ref/music.html`.

In the constructor of the `UserInterface` class, we add the following lines to get a piece of music playing at the beginning of the game:

```
pygame.mixer.music.load("17718_1462204250.mp3")
pygame.mixer.music.play(loops=-1)
```

Copy and paste these lines at the beginning of a Pygame program, update the music file, run, and listen!

Sound: contrary to music, we can load and store several sounds in Python variables using the `pygame.mixer.Sound` class constructor. For instance:

```
fireSound = pygame.mixer.Sound("fire.wav")
explosionSound = pygame.mixer.Sound("explosion.wav")
```

Then, we use the `play()` method of the `pygame.mixer.Sound` class to play the sound:

```
fireSound.play()
```

We can play several sounds at the same time. There are many other methods, like the `set_volume()` method that sets the volume of the sound. You can find more details here: `https://www.pygame.org/docs/ref/mixer.html`.

4.4.2 A naive approach

A naive way to play music or a sound is to repeat the two previous lines when something is modified. For instance, at the end of the `update()` method of the `PlayGameMode` class, when we evaluate if the game is over, we could immediately change the music:

```
if not self.playerUnit.alive:
    self.gameOver = True
    pygame.mixer.music.load("17675_1462199580.mp3")
    pygame.mixer.music.play(loops=-1)
```

It works fine, but we introduce a dependency between the game updates and the user interface. We don't like dependencies in software design and try to avoid them as much as possible.

Imagine that we want to create a multiplayer version of our game. In this case, there is a game server that runs the updates. This server has no screen and no user interface. With the naive approach, the server will play music! Even if it works fine, it uses CPU and disk I/O unnecessarily.

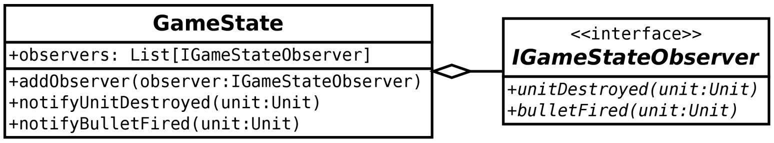
We could patch the code to add a condition to play music, like `if self.playMusic: ...`. It is a small example, but there are thousands of such triggers in real cases. We would lose a lot of time and risk forgetting to patch some lines.

They are also many other examples like this one, and as usual, I invite you to trust the many developers that faced them. They finally created solutions like the one we present in the next section.

4.4.3 Sounds with the Observer pattern

The Observer pattern allows us to remove the dependency between modules while creating links. These links are dynamics, and the observed one does not have to worry about who observes it. With this pattern, the game state can notify that something happens (a unit died, a bullet is fired). Then, anyone, including a sound layer, can react to these events.

Game state observer: in the current program, the game state can notify any observer when a unit is destroyed. We add a new `bulletFired()` method to our observer implementation to handle bullet shots (we only show related attributes and methods):


GameState
+observers: List[IGameStateObserver]
+addObserver(observer:IGameStateObserver)
+notifyUnitDestroyed(unit:Unit)
+notifyBulletFired(unit:Unit)
<<interface>>
IGameStateObserver
+unitDestroyed(unit:Unit)
+bulletFired(unit:Unit)

Then, in the `ShootCommand` class, we call the `notifyBulletFired()` method of the `GameState` class to indicate that we fire a bullet.

Sound layer: We create a new `SoundLayer` class, child of the `Layer` class, to play sounds:

```
1  class SoundLayer(Layer):
2      def __init__(self, fireFile, explosionFile):
3          self.fireSound = pygame.mixer.Sound(fireFile)
4          self.fireSound.set_volume(0.2)
5          self.explosionSound = pygame.mixer.Sound(explosionFile)
6          self.explosionSound.set_volume(0.2)
7      def unitDestroyed(self, unit):
8          self.explosionSound.play()
9      def bulletFired(self, unit):
10         self.fireSound.play()
11     def render(self, surface):
12         pass
```

Even if this layer does not render anything, we also consider it is contributing to the multimedia user experience. The implementation of this class is straightforward:

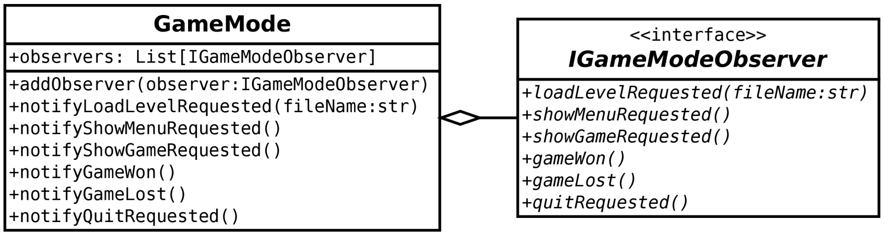
- It loads the sounds in the constructor (lines 3-6). Note that we choose to reduce the volume to put music in front. We could also add options in the menu to let the player set the volume;

- It plays the explosion sound when a unit dies (line 8);
- It plays the fire sound when a bullet is fired (line 10).

Pygame handles well the playing of many sounds. If it was not the case, we could handle it easily in this class, for instance, using a delay or a maximum number of simultaneous sounds. It would be easy because we use the Observer pattern. Otherwise, we would have to carry these values (delay or maximum) in many places in the program!

4.4.4 Music with the Observer pattern

Music changes are on a higher level than the game updates. For instance, it can happen between game creation and destruction. As a result, we apply the Observer pattern to the game modes (we only show related attributes and methods):


GameMode
+observers: List[IGameModeObserver]
+addObserver(observer:IGameModeObserver)
+notifyLoadLevelRequested(fileName:str)
+notifyShowMenuRequested()
+notifyShowGameRequested()
+notifyGameWon()
+notifyGameLost()
+notifyQuitRequested()
<<interface>>
IGameModeObserver
+loadLevelRequested(fileName:str)
+showMenuRequested()
+showGameRequested()
+gameWon()
+gameLost()
+quitRequested()

There are many cases to handle and as many methods. Don't be afraid of the number of these methods. There is software that automatically creates these methods and their implementation. If you don't want to use them or can't afford them, you can implement your processes. It is effortless with Python because the standard library already contains a code parser and updater that work very fine for refactoring.

Notifications: there are two categories of notifications: events and requests. The events indicate that we changed something, like `gameWon()`. The requests are queries: maybe we didn't change anything, but we would like something to happen, and we can't do it ourselves. For instance, `showMenuRequested()` means that we would like that the menu pops up, but we can't do it. For instance, the game runs, and the player hits the escape key.

When the game is running, the `PlayGameMode` is active and captures the key events. The player hits the escape key, meaning that (s)he wants to get the menu. The `PlayGameMode` class must not handle that because it involves a higher level in the user interface (it destroys and creates modes, including the play game mode). We already delegated this to the `UserInterface` class thanks to its `showMenu()` method in the previous program.

Calling a method of the `UserInterface` class in the `PlayGameMode` creates a circular dependency. Instead, the `PlayGameMode` calls the `notifyShowMenuRequested()`

method. If UserInterface observes PlayGameMode, it can show the menu when we make this request.

Implementation: in many cases, the implementation is straightforward: we replace the call to a method of the UserInterface by a notification. For instance, in the processInput() method of the MessageGameMode class, we replace self.ui.quitGame() by self.notifyQuitRequested() and self.ui.showMenu() by self.notifyShowMenuRequested()

```
for event in pygame.event.get():
    if event.type == pygame.QUIT:
        self.notifyQuitRequested()
        break
    elif event.type == pygame.KEYDOWN:
        if event.key == pygame.K_ESCAPE \
        or event.key == pygame.K_SPACE \
        or event.key == pygame.K_RETURN:
            self.notifyShowMenuRequested()
```

GameMode child classes no more have a ui attribute that points to the UserInterface class. Therefore, we remove the dependency.

For the other cases, where something happens, we update the user interface and change the music when necessary. For instance, when the game is lost or won:

```
def gameWon(self):
    self.showMessage("Victory !")
    pygame.mixer.music.load("17382_1461858477.mp3")
    pygame.mixer.music.play(loops=-1)

def gameLost(self):
    self.showMessage("GAME OVER")
    pygame.mixer.music.load("17675_1462199580.mp3")
    pygame.mixer.music.play(loops=-1)
```

4.4.5 What to remember

Identify and remove circular dependencies: we can not emphasize this enough: find circular dependencies and remove them. For example, if two classes call methods from each other, we get a case.

A solution is to see if there are ways to remove calls from one side. For example, sometimes, we call methods because we need data from the other class. In this case, we can get this data and store it in class attributes.

Another solution is to use the Observer pattern. One of the two classes observes the other, which removes the dependencies to the implementation. The only object on which both classes depend is the observer interface, a list of method and argument names, and no implementation.

Play music: the `pygame.mixer.music.load()` function loads a music:

```
pygame.mixer.music.load("music.mp3")
```

We can only have one music at a time. To play the music, we use the `pygame.mixer.music.play()` function:

```
pygame.mixer.music.play(loops=-1)
```

The `loops` argument is the number of times Pygame should play the music. With a value of `-1`, it repeats indefinitely.

Play sound: we load a sound file when we create instances of the `pygame.mixer.Sound` class:

```
sound = pygame.mixer.Sound("sound.wav")
```

We can load as many sound files as we want. To play a sound, use the `play()` method:

```
sound.play()
```

We can play several sounds at the same time.

4.5 Game modding

Today, players can easily modify their games thanks to tools provided by game developers. We see in this section how we can create such helpers for the assets of the tank game.

Although we make these tools for players, they are also interesting for us: they ease the creation of new game content. As a result, it is common in the game industry to create these extra features for internal use.

4.5.1 Describe assets in a file

Create the file: we gather in a JSON file the information on the game assets (we don't present all of them, only samples of each category):

```json
{
  "font": {
    "title": "BD_Cartoon_Shout.ttf",
    "titleSize": 64
  },
  "tile": {
    "width": 64,
    "height": 64,
    "ground": "ground.png"
  },
  "sound": {
    "fire": "170274__knova__rifle-fire-synthetic.wav"
  },
  "music": {
    "play": "17687_1462199612.mp3"
  }
}
```

The JSON format is a format that anyone can read and modify, contrary to binary formats. We choose this one because the standard Python library contains a parser. The XML format is also part of the standard library, but the syntax is more complicated. The YAML format is perfect for configuration files (more compact) but requires an additional package. We could also use Python to describe our assets; however, it is dangerous because somebody could introduce malicious code.

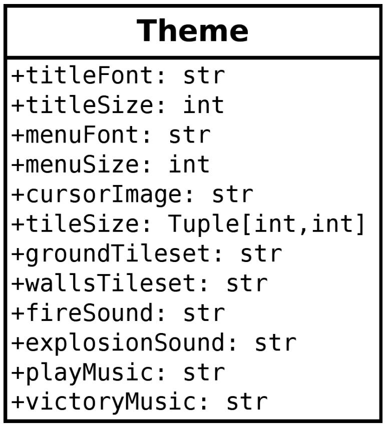
Parse the file: whatever the format (JSON, XML, or YAML), we create a data tree with string keys and values as strings or numbers. Then using a parser, we get a similar tree but in Python. For JSON, the parsing of a file is straightforward:

```python
import json
with open("theme.json", encoding='utf-8') as file:
    data = json.load(file)
```

The `data` variable is a dictionary with the same keys as in the JSON tree. For instance, `data["font"]` gives access to the font assets. We can also stack the keys, for instance, to get the font title:

```
titleFont = data["font"]["title"]
```

4.5.2 Parse and store the data



```
Theme
+titleFont: str
+titleSize: int
+menuFont: str
+menuSize: int
+cursorImage: str
+tileSize: Tuple[int,int]
+groundTileset: str
+wallsTileset: str
+fireSound: str
+explosionSound: str
+playMusic: str
+victoryMusic: str
```

We define a new `Theme` class that parses and stores all our assets' information (NB: in the figure, we don't show all properties). We use a reference to an instance of this class in all classes that render (`UserInterface`, children of `Layer`, etc.). Then, they can access any theme value (sound file, tile size, etc.).

Parse the data: as we saw before, we should do as many checks as possible when reading data from a file. For instance, we check that the file exists for strings related to a file. In this area, a naive approach consists in repeating the checks for every case. I hope you start to know the story: we don't like to write similar code several times; it takes time and is error-prone.

Consequently, we define a function that does the checks:

```
def failIfNotExists(data, section, name):
    if section not in data:
        raise RuntimeError(
            "No section {} in {}".format(section, file))
    data = data[section]
    if name not in data:
        raise RuntimeError(
            "No section {}.{} in {}".format(section, name, file))
    fileName = data[name]
    if not os.path.exists(fileName):
        raise RuntimeError("No file {}".format(fileName))
    return fileName
self.titleFont = failIfNotExists(data, "font", "title")
self.menuFont = failIfNotExists(data, "font", "menu")
```

First of all, note that we can define a function inside a function (or a method); in this case, we define this function in the constructor of the `Theme` class. Using this feature of the Python language, we don't have to worry about the name of the function: another one with the same name can exist elsewhere, there is no conflict.

The `failIfNotExists()` function has three arguments: the data extracted from the JSON file, the key in the first level of the data tree, and the key in the second level. We expect the value to be in `data[section][name]`. However, if there is an error in the file or the expected data is missing, Python will raise exceptions hard to understand for a user. So, we run tests and build messages that better help.

The function checks that the key exists in the dictionary (lines 2-8). The syntax is `"key" not in dictionary`, which returns `True` if the key “key” is *not* in the dictionary (remove the `not` keyword if you want to check if the key is in the dictionary).

Finally, the function ensures that the file exists using the `os.path.exists()` function (line 10), which returns `True` if the file exists.

4.5.3 Optional assets

We should always try to have something that runs; every time the player can’t play the game because of a problem, we lower its experience. Considering assets, if some of them can be missing, we should validate the data and run the game without it.

It is the case for sounds and music: we can play the game without them. So, we create another function to check the data, and if there is an issue, we return `None` (no exception):

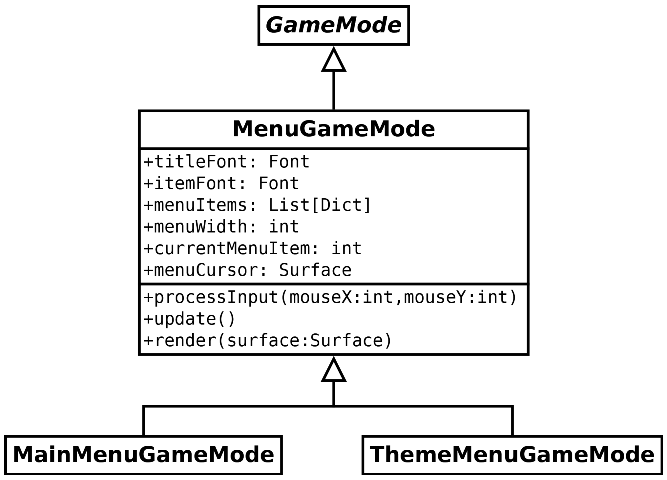
```
1  def setIfExists(data, section, name):
2      if section not in data:
3          return None
4      data = data[section]
5      if name not in data:
6          return None
7      fileName = data[name]
8      return fileName if os.path.exists(fileName) else None
9  self.fireSound = setIfExists(data, "sound", "fire")
10 self.explosionSound = setIfExists(data, "sound", "explosion")
11 ...
```

Note line 8: we use a compact syntax that can return two values, depending on a condition. It is equivalent to:

```
if os.path.exists(fileName):
    result = fileName
else:
    result = None
return result
```

4.5.4 Create a theme menu

One menu implementation: we created theme files, and we want the player to select one from the game menu. We could copy and paste the `MenuGameMode` class and only change the menu items. Instead, we remove the items' definitions for `MenuGameMode` and create child classes for each menu. For instance, `MainMenuGameMode` for the main menu and `ThemeMenuGameMode` for selecting a theme:



Static menu: we give a list of menu items in the constructor (as before):

```python
class MainMenuGameMode(MenuGameMode):
    def __init__(self, theme):
        menuItems = [
            {
                'title': 'Play',
                'action': lambda: self.notifyShowMenuRequested("play")
            },
            ...
        ]
        super().__init__(theme, menuItems)
```

Note that we can run code before calling the base constructor (e.g. `super().__init__()`).

Theme menu: for the theme selection menu, we search for ".json" files in the current directory. For each file we find, we create a menu item:

```python
class ThemeMenuGameMode(MenuGameMode):
    def __init__(self, theme):
        menuItems = []
        for file in os.listdir("."):
            name, ext = os.path.splitext(file)
```

```
 6              if ext != ".json":
 7                  continue
 8              menuItems.append({
 9                  'title': name,
10                  'action': lambda file=file:
11                                self.notifyChangeThemeRequested(file)
12              })
13          menuItems.append({
14              'title': 'Back',
15              'action': lambda: self.notifyShowMenuRequested("main")
16          })
17          super().__init__(theme, menuItems)
```

The `os.listdir()` function returns a list of files found in a directory. In our case, we look in the current directory "." (line 4). We use the `os.path.splitext()` to get the file name and extension ("file.json" returns "file" and ".json"). We ignore files that are not JSON files (lines 6-7).

Lines 8-12 add a new item to the list. As before, we build a dictionary with two elements, one for the title and one for the action. Note the lambda expression:

```
lambda file=file: self.notifyChangeThemeRequested(file)
```

We add an argument "file" with a default value "file". As for any function, this is not the same variable. The former is the first argument of the lambda (used inside the lambda); the latter is the variable in the constructor (created in the `for` statement).

We ensure that each lambda function has a different file using this trick. If we remove the argument of the lambda `file=file`, all lambdas will have the same file (the last of the loop). It is because lambda captures the variables in their context. So, for example, the lambda knowns `self`, which points to the instance of `ThemeMenuGameMode`, even if we don't declare any argument that copies it. It is the same for `file`: only one variable with this name is in the constructor, and consequently, only one value.

Other improvements: we implement a new menu for level selection in the same way: we look for ".tmx" files in the current directory and create a menu item for each case. There are also some changes to manage several menus: you can read these updates in the code attached to this section.

4.5.5 What to remember

Define a function inside a function: we can define a function inside a function or a method. It reduces the code length and increases readability. Furthermore, we don't have name conflict since the nested function's name is limited to its context.

Check if an item is in a container: we can use the following syntax to know if a value is inside a container (list, dictionary, etc.):

```
value in container
```

Compact if/else (ternary conditional operator): we can select between two values with the following syntax:

```
result = value1 if condition else value2
```

It is equivalent to:

```
if condition:
    result = value1
else
    result = value2
```

File/directory operations:

- `os.path.splitext()`: returns the base name and extension of a file;
- `os.path.exits()`: returns `True` if the file exists;
- `os.listdir()`: returns files of a directory.

Lambda function in loops: this is a typical error that many programmers make. In a context where a variable content changes, we must copy its value if we want to use it in lambda functions:

```
functions = []
for i in range(6):
    functions.append(lambda i=i: 2*i)
for function in functions:
    print(function())
```

The trick is the argument `i=i` is the lambda. It ensures that each lambda function has a unique value. Otherwise, all lambdas will have the same value (the last one). Run the code with and without the `i=i` to see the difference.

4.6 Modules and packages

For the sake of simplicity, we coded our game in a single file. Unfortunately, as the number of lines increases, it becomes difficult to read. This section creates modules and packages to split the code into multiple files and directories.

4.6.1 Python modules

When creating a Python ".py" file, we automatically define a new module. A module is a collection of Python elements like variables, functions, or classes.

For instance, we create the following "functions.py":

```python
# Content of functions.py
def f(x):
    return 2 * x
```

If we run "functions.py", it does nothing.

We also create a file "run.py":

```python
# Content of run.py
print(f(2))
```

If we run "run.py", there is an error: "NameError: name 'f' is not defined". It is as expected: modules are disconnected, Python won't automatically look in all files to find the missing function.

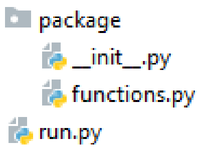
If we add a `from ... import` statement as we did with library packages, there is no error, and the console shows the correct result:

```python
# Content of run.py
from functions import f
print(f(2))
```

Note the two parameters in the `from ... import` statement: the first is the module/file name without ".py", and the second is the function we want to import.

4.6.2 Python packages

Create a package: we can gather modules/python files into a directory. Then, if we add an “**init**.py” file in this directory, the directory is a package. For instance, if we create a directory “package” with an “**init**.py”, and move the “functions.py” in this directory:



Then, we can access the `f` function with the following syntax:

```
# Content of run.py
from packages.functions import f
print(f(2))
```

The “`__init__.py`” file does not need to have content. However, we can add code in this file; Python executes it when someone imports it. So, for example, if we add `print("import package")` in the “`__init__.py`” file, we see the message before the result of the function computed in “run.py”.

Relative import: the default import links are absolute: they start from all the directories in the python path, including the running directory. We can use relative links when we need a module next to the current one. For instance, if we create a “functions2.py” next to “functions.py”:

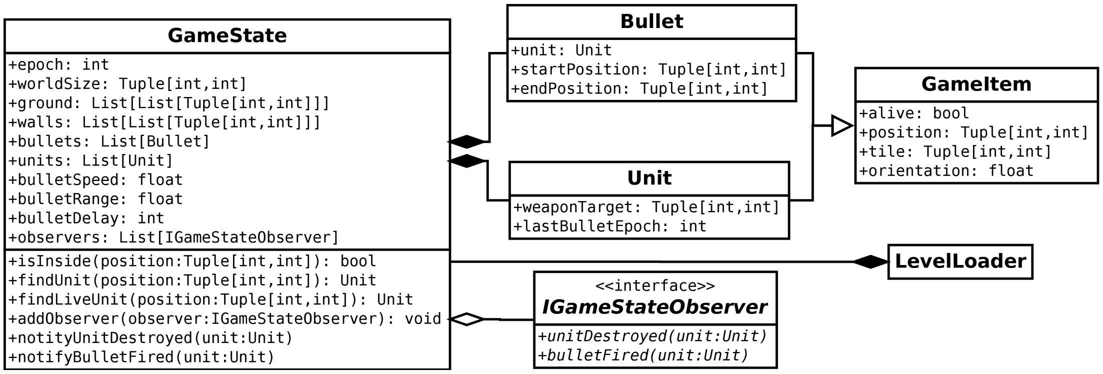
```
# Content of functions.py
from .functions2 import g
def f(x):
    return 3 * g(x)

# Content of functions2.py
def g(x):
    return 2 * x
```

The statement `from .functions2 import g` means “import `g` from the functions.py in the same directory”.

We can split our code into multiple directories/packages and files/modules thanks to these new features.

4.6.3 Game state package

We create a directory “state” with an “`__init__.py`”. Then, we add a python file for each class related to the game state, where each file has the name of the class:


GameState
+epoch: int
+worldSize: Tuple[int,int]
+ground: List[List[Tuple[int,int]]]
+walls: List[List[Tuple[int,int]]]
+bullets: List[Bullet]
+units: List[Unit]
+bulletSpeed: float
+bulletRange: float
+bulletDelay: int
+observers: List[IGameStateObserver]
+isInside(position:Tuple[int,int]): bool
+findUnit(position:Tuple[int,int]): Unit
+findLiveUnit(position:Tuple[int,int]): Unit
+addObserver(observer:IGameStateObserver): void
+notityUnitDestroyed(unit:Unit)
+notifyBulletFired(unit:Unit)
Bullet
+unit: Unit
+startPosition: Tuple[int,int]
+endPosition: Tuple[int,int]
GameItem
+alive: bool
+position: Tuple[int,int]
+tile: Tuple[int,int]
+orientation: float
Unit
+weaponTarget: Tuple[int,int]
+lastBulletEpoch: int
LevelLoader
<<interface>>
IGameStateObserver
+unitDestroyed(unit:Unit)
+bulletFired(unit:Unit)

These classes have no dependency outside the package, except for library packages (os and tmx). In the following, when we say that a class has no dependency, it will mean that there is no dependency inside our code.

Some classes depend on another one: we represent these dependencies with a line and a symbol in the diagram.

Inheritance: `Unit` is a child class of `GameItem`, represented with a white arrow in the diagram. In the python file, we have to import the superclass:

```python
from .GameItem import GameItem
class Unit(GameItem):
    ...
```

We use a relative import; the absolute equivalent is `from state.GameItem import GameItem`. Also, note that we repeat the name `GameItem` twice in the import statement. However, these two names do not refer to the same object: the first one (after `from`) is the module/python file, and the second one (after `import`) is the class.

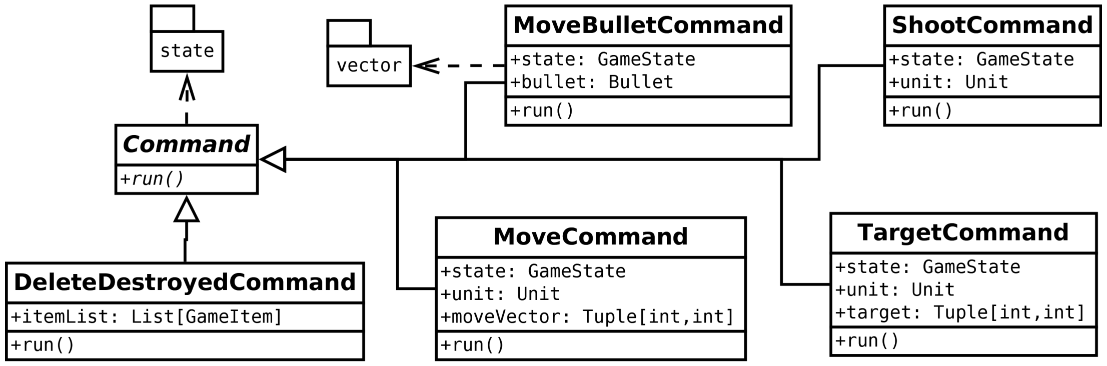
Composition: some classes contain instances of others. For instance, `GameState` holds instances of `Unit` and `Bullet`. On the diagram, we represent it with a black diamond.

Note that a Python object can’t contain another object; it must be a reference. So, when we say that a class “contains” instances, it is a design choice. It means that the life of the container and the object are bound. If we destroy one, we must destroy the other. Do not worry if this is not clear; we will see examples in the next chapters.

Aggregation: some classes contain references to others, represented with a white diamond. This time, we consider references, meaning that the life of objects is not bound: if we destroy the container, it does not delete the object. In the case of the game observers, if we delete the game state, we don't have to delete the observers and vise-versa.

4.6.4 Command package

We create a new `command` package, and add all the command classes:



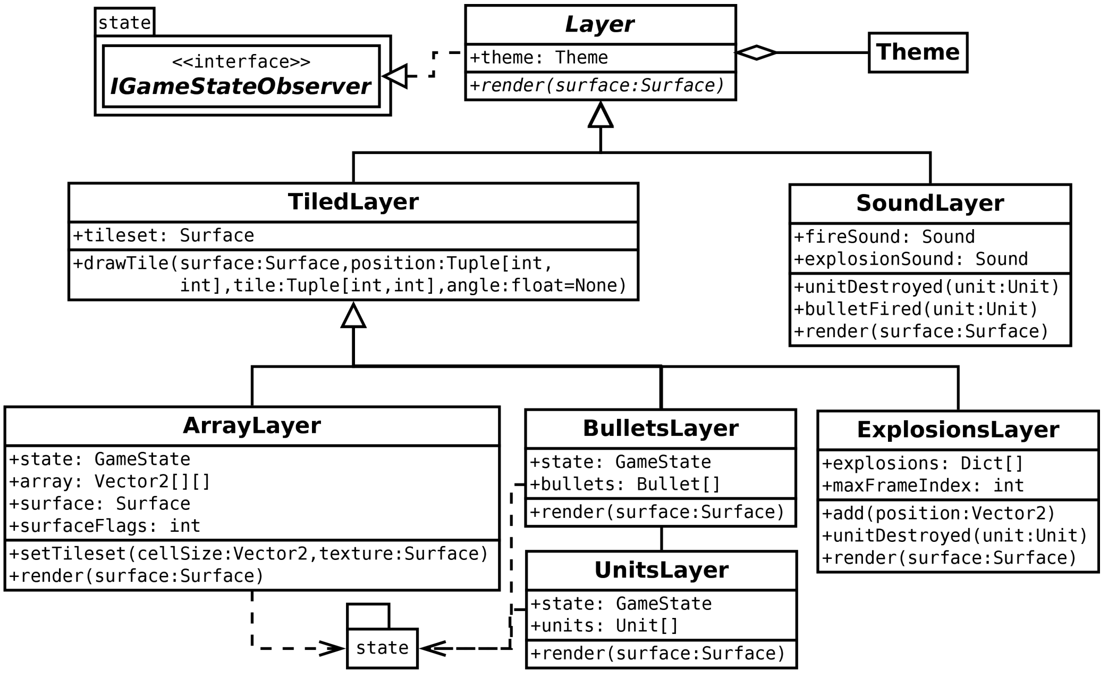
Note the `state` component in the diagram: it represents the `state` package. Furthermore, there is a dashed arrow from the `Command` class to `state`: it means that the class depends on the package. It is a general dependency; the component can use many items in many ways in the package.

There is also a dependency on a `vector` package. It is a small package where we put all functions on vector computations (distance, norm, etc.). We also need them in other packages, and as usual, we don't want to have multiple times the same code. Furthermore, we don't put these functions in this package since we could introduce unnecessary dependencies.

Arrows in these diagrams are helpful to check dependencies. In the best case, we want to find a tree, or, in other words, a graph with no cycles (remember: we hate circular dependencies!). In this example, we can easily see the tree: the `state` package is the root, `Command` is a node, and all child classes are leaves.

4.6.5 Layer package

The `layer` package contains all layer classes:



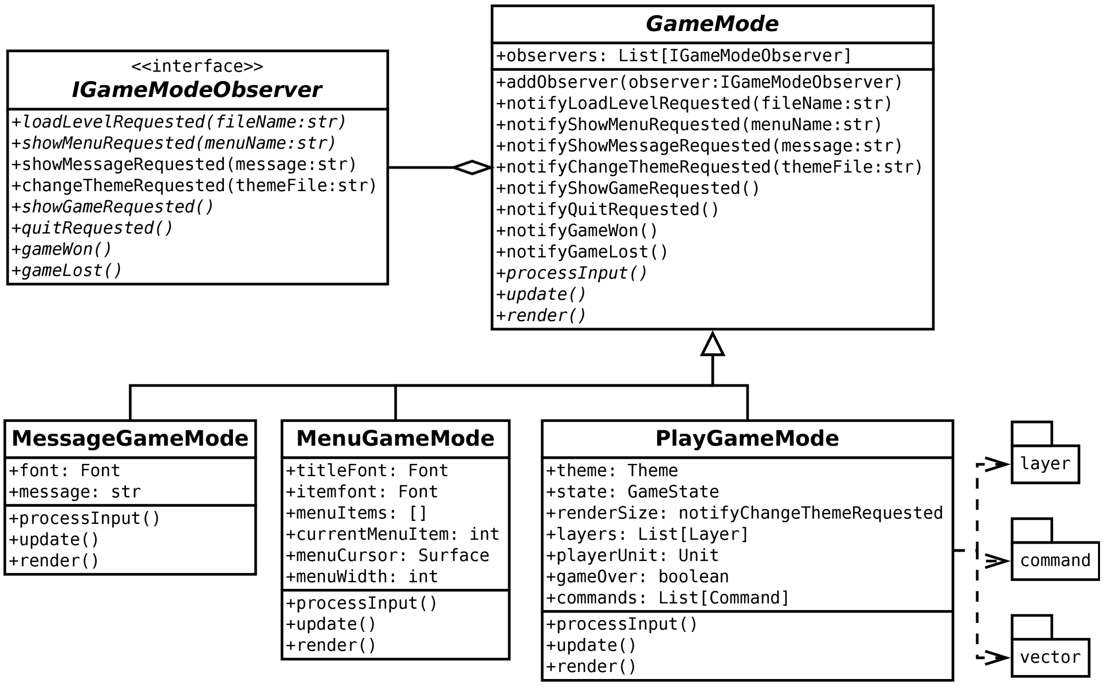
There are two external dependencies: one on the `state` package and the other one on the `IGameStateObserver` interface in the same package.

The first dependency is general. We can see that only three classes depend on `state`; it could be interesting in some cases.

The second dependency emphasis an interface. Interface only contains methods, and most of the time, they have no dependencies. It means that the `Layer` class has nearly no dependency on the `state` package (and consequently, some of its children, like `SoundLayer`).

4.6.6 Game mode package

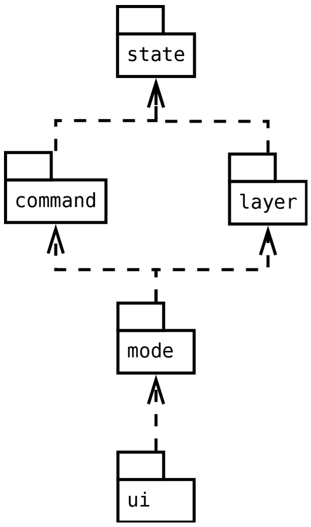
The `mode` package contains our modes (we don't show the children of `MenuGameMode`):



At a glance, we can see that only the `PlayGameMode` class depends on other packages but not the other modes. It is the magic of the Observer pattern.

For instance, when the level menu needs to load, it doesn't have to create all the required components (rendering, state, logic, etc.). Instead, it only notifies that a level has to be loaded; if someone is listening, then the game will start. If no one is listening, that is not the problem of the menu.

4.6.7 The big picture



Finally, we put the `UserInterface` class in the `ui` package. It is the only class: the job of the tank game UI is simple enough to fit in some methods. The figure shows a diagram with all our packages and their dependencies (we don't show the `vector` package because it acts as an external library).

We can see a hierarchy in our dependencies: there are no cycles. The `state` component is the root of this hierarchy. It does not depend on other components. So, we can delete everything else; it still works.

Then, the `command` and `layer` components are the first nodes. They are independent: we can create one without the other. However, we must have the `state` component; otherwise, they can't work.

The `mode` component depends on the `command` and `layer` components: if one is missing, we can't use it. Since these two dependencies depend on `state`, it means that `mode` also depends on `state`. We don't draw an arrow between `mode` and `state` because we can easily see it: just follow the direction of arrows from `mode` to `state`.

The `ui` component depends on the `mode` component, and consequently, on all the components of this architecture.

4.6.8 What to remember

Create a package: create a directory with a "`__init__.py`" file inside.

Relative import: we can import a component from the same directory with the following syntax:

```
from .Module import Component
```

Dependencies in diagrams:

- White arrow: inheritance;
- Black diamond: composition, one object contains the other; the two objects are bound: if we delete one, we must delete the other;
- White diamond: aggregation, one object contains a reference to the other; the two objects are not bound;
- Dashed arrow: other cases, general dependency.

Index