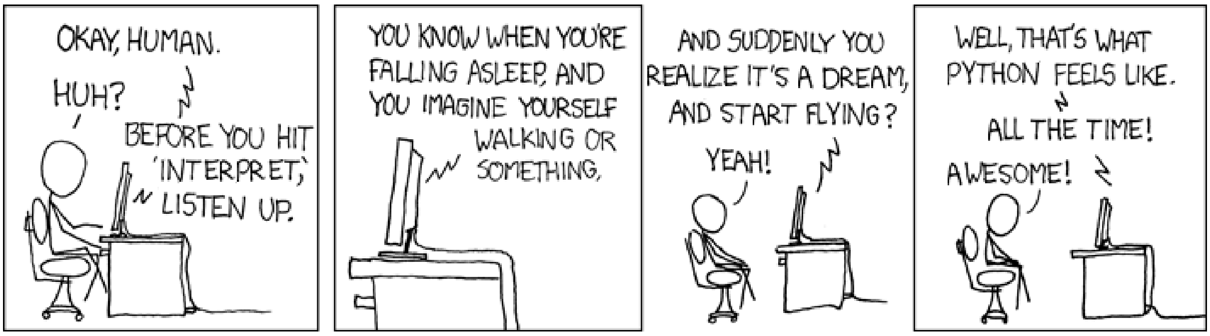
Digitmind in Python

A 🐍 *tale...* 😎

Preface



In 2017 the scripting language Python rose to the 1st place of the IEEE Spectrum ranking of the top programming languages[1]. One of the major drivers of this is the fact that Python is the language of choice in most Deep Learning programmes. This is not in the least because of the use of Python as part of Google's Deep Learning API TensorFlow. So, as stated on TensorFlow's getting-started page[2], nowadays it's preferable if:

> "*You can code (at least a little) in Python.*"

When learning a new programming language I always find it helpful to go beyond the obligatory "Hello world"[3] and program the game Mastermind. To date I've done this for Extended Basic, Turbo Pascal, Delphi, C/C++, COBOL and Objective-C.

For Python I documented this process. As such this document contains the software design of a *Digitmind* game written in Python. The game is based on MasterMind, but instead of colors one has to guess a code of 4 different digits.

Focusing on core mechanics, the program uses the command line, so no GUI design rules were hurt during this process…

By the way - for convenience the final program is available on GitHub.

My goal with this document is also to give the novice Python programmer a fun introduction into some of the powerful features of the language.

February 2018, Fred Dijkstra 🤓

[1] https://spectrum.ieee.org/computing/software/the-2017-top-programming-languages
[2] https://www.tensorflow.org/get_started/get_started_for_beginners
[3] https://en.wikipedia.org/wiki/%22Hello,_World!%22_program

Table of contents

Introduction

This document contains the software design of the *Digitmind* game written in Python. The game is based on the well known *MasterMind*, but instead of colors one has to guess a code of 4 *different* digits.

The difficulty level of DigitMind can be set by specifying the number of digits:

1. [1,2,3,4]
2. [1,2,3,4,5]
3. [1,2,3,4,5,6]
4. [1,2,3,4,5,6.7]
5. [1,2,3,4,5,6,7,8]
6. [1,2,3,4,5,6,7,8,9]
7. [1,2,3,4,5,6,7,8,9,0]

A score is defined as two numbers:

- *Correct position*; i.e. the number of digits that are part of the code and are at the correct position.
- *Wrong position*; i.e. the number of digits that are part of the code, but placed at the wrong position.

Using the scores, a player has to guess the code. This will be quite a challenge at the higher levels!

In the next chapter we'll first determine how to create a function that can calculate the score. In the subsequent chapters, this function will then be used to implement an algorithm that enables the computer to become a codebreaker and to give the score when a human plays the role of codebreaker.

Scoring

The score comprises two values: the number of digits on the right position and the number of digits on the wrong position. In this chapter, two possible implementations of a function are presented.

Method

To hold the score, a dictionary is used:

```
score = {'correct position':0, 'wrong position':0}
```

To determine this score, a function is created that takes two combinations and determines the score and returns this dictionary:

```
def determine_score(guess, code)
```

A straightforward implementing this function is:

- Loop through the elements of the **`combination`** and check if the digit at the same position in the **`code`** is the same.
- If so, increment **`score['correct position']`**.
- If not, then check whether the digit is present in the code.
- If so, then increment **`score['wrong position']`**.

In the following function this is implemented:

```
def determine_score(guess, code):
    score = {'correct position':0, 'wrong position':0}

    for i in range(len(guess)):
        if guess[i] == code[i]:
            score['correct position'] += 1
        elif guess[i] in code:
            score['wrong position'] += 1

    return score
```

This implementation is straightforward and can for example be understood by any regular C-programmer with no Python experience. However, it is not really 'Python-like'. There is also an alternative as discussed in the following section.

Alternative implementation

An alternative implementation - of which one could say is more *Pythonic* - uses built-in Python functions. This alternative implementation comprises two steps:

1. Create pairs out of both lists and count the number of pairs that are the same.
2. Convert the lists to sets, create the *intersection*, which determines the number of elements that are present in both lists. Determine the length of this intersection and subtract the number of elements that are in the same position, as determined in the previous step.

To perform step 1, the built-in `zip()` function can be used. As an example, consider the following:

```
guess = [9,8,7,6]
code  = [1,8,7,4]
pairwise = list(zip(guess, code))
```

This results in **`pairwise`** being equal to **`[(9,1),(8,8),(7,7),(6,4)]`**.

This can now be used in a list comprehension in combination with the `sum()` function:

```
guess = [9,8,7,6]
code  = [1,8,7,9]
pairwise = zip(guess, code)
num_matched_digits = sum(1 for p in pairwise if p[0] == p[1])
```

*Note: to use **`pairwise`** it does not have to be converted to a list as it is already an iterable object.*

In this example the result **`num_matched_digits`** is equal to 2 because **`guess[1] == code[1]`** and **`guess[2] == code[2]`**.

To determine the number of elements in the list that are the same, but not necessarily in the same position, the `set.intersection()` function can be used. Consider the example:

```
a = set(guess).intersection(set(code))
print(a)
```

This prints the set **`{8,9,7}`** as these are the digits that are present in *both* lists. The length of this minus the value of **`num_matched_digits`** in the previous example will now yield the number of digits that are present in both lists but not in the correct position.

All this can now be used in a function:

```
def determine_score_alternative(guess, code):
    pairwise = zip(guess, code)
    score['correct position'] = sum(1 for p in pairwise if p[0] == p[1])
    score['wrong position']   = len(set(guess).intersection(set(code))) - \
                                score['correct position']
    return score
```


Comparison

It can be argued that which implementation is preferable also depends on style, but the *Pythonic* way is more concise. I personally especially really like the statement:

```
score['correct position'] = sum(1 for p in pairwise if p[0] == p[1])
```

However, for sure that implementation takes more time to explain when the concepts behind the **`zip()`** and **`set.intersection()`** functions are new to a novice Python programmer.

I've done some performance comparisons and the results vary a little, but the alternative implementation seems a small fraction faster, so we'll go for that one.

Computer codebreaker

When the computer plays the role of codebreaker, we must choose a code of 4 different digits and let the computer guess the correct combination.

Algorithm

So, the question is, what algorithm do we need to implement to make it possible for the computer to break the code? For our demonstrator we're going to use a fairly simple algorithm:

```
Specify the difficulty_level;
Create all possible combinations;
Set score to its initial value;
while not all digits match do:
    Randomly choose guess from remaining combinations;
    Get score for guess;
    Process score by removing all non-matching combinations;
end while;
```

We will describe these steps in the following sections.

Specifying the difficulty level

To specify the difficulty level, we need the keyboard **`input()`** function:

```
difficulty_level = int(input("Give the difficulty level [1..7]: "))
```

Note that we need to convert the result to an integer.

Creating the combinations

The possible combinations of 4 different digits depends on the difficulty level which specifies the number of digits by: **`difficulty_level+3`**.

Using list comprehensions, a function can be created to construct the list of combinations:

```
def create_combinations(difficulty_level):
    digits = [1,2,3,4,5,6,7,8,9,0][:difficulty_level+3]
    return [(w,x,y,z)           \
        for w in digits         \
        for x in digits         \
        for y in digits         \
        for z in digits         \
        if  w not in (x,y,z)    \
        and x not in (y,z)      \
        and y != z ]
```

Note that the combination **`(w,x,y,z)`** is defined as a tuple, which is a sequence of immutable Python objects. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.

Also note the concise way of defining the number of digits, given the difficulty level:

```
digits = [1,2,3,4,5,6,7,8,9,0][:difficulty_level+3]
```

This statement actually performs the following steps:

1. Create a list holding all the digits, with the zero defined as the last digit.
2. Only take the *first* **`difficulty_level+3`** elements of the created list.
3. Assign the resulting list to the **`digits`** variable.

For a difficulty level of 7, this obviously means all 10 digits, in which case the **`create_combinations()`** function will create 10x9x8x7 = 5040 combinations.

Making a guess

To make a new guess, the computer simply selects a random combination out of this list of remaining combinations. To do this, the random module contains a **`choice()`** function that can be used:

```
import random
# ...
combinations = create_combinations(difficulty_level)
guess = random.choice(combinations)
```

Processing a score

Given a score, the possible combinations can be reduced. This is done by looping through all combinations and removing those that would give a *different* score.

Removing an element from a list at a specific index can be done using the **`del`** statement which takes the *element* to be deleted. As an example:

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

This will result in **`[1,2,4]`** as the third element **`x[2]`** is deleted.

The problem here now is that while looping through the list, elements are removed from it. This means that we need to loop *backwards* through the list. This can be done by using **`reversed`** to get a *reverse iterator* of the list in combination with **`enumerate`** to get the index of the current element. As an example, the following code removes all the odd numbers from a :

```
x = list(range(10)) # Creates a list of integers from 0 to 9
max_index = len(x)-1

for i, v in enumerate(reversed(x)):
    if v % 2:
        del x[max_index-i]

print(x)
```

We can now use this to build a function that processes the score:

```
def process_score(combinations_left, tried_combination, score):
    max_index = len(combinations_left)-1
    for i, combination in enumerate(reversed(combinations_left)):
        if score != determine_score(combination, tried_combination):
            del combinations_left[max_index-i]
```

Computer move

When the computer needs to make a move, it can randomly chooses one element from the remaining combinations and asks for the score:

```
def do_computer_move(combinations_left):
    guess = random.choice(combinations_left)

    print('My guess:', guess)

    score['correct position'] = int(input('How many are in the correct position? '))

    if score['correct position'] != 4:
        score['wrong position'] = int(input('How many are in the wrong position? '))
        process_score(combinations_left, guess, score)

    return score
```

The computer keeps making a move until it has 4 digits on the right position:

```
score = {'correct position':0, 'wrong position':0}
while score['correct position'] != 4:
    score = do_computer_move(combinations, difficulty_level)
print('YES!')
```

For a code of `[1,3,4,9]` this can result in the following flow:

```
My guess: (5, 1, 8, 9)
How many are in the correct position? 1
How many are in the wrong position? 1
My guess: (7, 2, 8, 1)
How many are in the correct position? 0
How many are in the wrong position? 1
My guess: (2, 3, 5, 9)
How many are in the correct position? 2
How many are in the wrong position? 0
My guess: (0, 3, 1, 9)
How many are in the correct position? 2
How many are in the wrong position? 1
My guess: (1, 3, 6, 9)
How many are in the correct position? 3
How many are in the wrong position? 0
My guess: (1, 3, 4, 9)
How many are in the correct position? 4
YES!
```

This corresponds to theory: for a combination of 4 different digits and given a correct score input for every guess, the maximum number of guesses is 6. Any lower number of guesses is due to luck.

Score input error handling

As stated, the score must be correct. So, how to detect an error in the score input?

When the computer is playing the role of codebreaker, the human player will need to specify the score. Obviously, errors can be made here. The result of an error will be that the list of combinations left will become empty at some point.

This can be used to detect the input error. In first instance, adding the following at the beginning of the **`do_computer_move()`** function seems like the way to go:

```
def do_computer_move(combinations_left, difficulty_level):

    # Input error handling
    if len(combinations_left) == 0:
        print("You probably made an error in specifying the score for one of my guesses!")
        print("Let's start over..")
        combinations_left = create_combinations(difficulty_level) # Error: Breaks reference!

    # ...
```

However, this code is incorrect: the **`combinations_left`** parameter of the function is supplied by reference. The assignment operation breaks this!

This is an important concept in Python and can be a source of frustration for the novice Python programmer.

Therefore, instead of the assignment operator, the **`extend()`** function is used since the array is empty anyway:

```
def do_computer_move(combinations_left, difficulty_level):

    # Input error handling
    if len(combinations_left) == 0:
        print("You probably made an error in specifying the score for one of my guesses!")
        print("Let's start over..")
        combinations_left.extend(create_combinations(difficulty_level))

    # ...
```

Human codebreaker

With the work done in the previous chapters, all necessary functions are now available to implement the functionality for a human to play the role of codebreaker.

For the human player to input the guess the **`input()`** function is used again. As such the guess will be input as a string, e.g. "1234". With one statement using list comprehension this can be converted to a list of integers:

```
guess = [int(c) for c in input("Your next guess: ")]
```

Using this:

```
combinations = create_combinations(difficulty_level)
code = random.choice(combinations)

print("Ok, I've chosen a code, try to guess it!")

score = {'correct position':0, 'wrong position':0}
while score['correct position'] != 4:
    guess = [int(c) for c in input("Your next guess: ") ]
    score = determine_score(code, guess)
    print(score)

print("\nCorrect! Well done!")
```

Below a result of an experienced Digitminder[4]:

```
Ok, I've chosen a code, try to guess it!
Your next guess: 1234
{'correct position': 2, 'wrong position': 0}
Your next guess: 1256
{'correct position': 1, 'wrong position': 0}
Your next guess: 1738
{'correct position': 0, 'wrong position': 2}
Your next guess: 7284

Correct! Well done!
```

[4] That would be the author 🤓

Alternative scoring type

The numerical nature of Digitmind makes it possible to consider a different way of scoring.

Method

For those people finding the conventional scoring of Mastermind too easy, the numerical nature of Digitmind makes it possible to consider a different scoring method in which for each position the absolute difference (for example 3-8=5) is determined between the digits in the guess and those in the code. These are then summed, yielding the score as implemented in a neat Pythonic one-liner:

```
def determine_score(combination, code):
    return sum(abs(combination[i] - code[i]) for i in range(len(combination)))
```

To incorporate this in our program, we need to do a little bit of reorganization of the code structure and are going to use an object oriented approach and introduce a **`ScoreCalculator`** class.

Score calculator

Before adding the functionality to use the different scoring method, let's first change the current program by add the **`ScoreCalculator`** class that has a method to determine the score and make the processing function (see "*Processing a score*") a method of the class as well:

```
class ScoreCalculator:

    def determine_score(self, guess, code):
        pairwise = zip(guess, code)
        self.score['correct position'] = sum(1 for p in pairwise if p[0] == p[1])
        self.score['wrong position'] =  len(set(guess).intersection(set(code))) - \
                                        self.score['correct position']
        return self.score

    def process_score(self, combinations_left, guess):
        max_index = len(combinations_left)-1
        for i, combination in enumerate(reversed(combinations_left)):
            if self.score != self.determine_score(combination, guess):
                del combinations_left[max_index-i]
```

This can now be used in the rest of the program. To do this, let's first create an instance of the **`ScoreCalculator`**:

```
# Instantiate score calculator
score_calculator = ScoreCalculator()
```

Next, the statement:

```
score = determine_score(code, guess)
```

needs to be changed everywhere to:

```
score = score_calculator.determine_score(code, guess)
```

Score format abstraction

For the alternative scoring method, the score will be a single integer. This means that the following statements must be changed:

```
score = {'correct position':0, 'wrong position':0}
while score['correct position'] != 4:
# ...
```

The first statement is the initialization of the **`score`**. If we simply let the **`ScoreCalculator`** be responsible for holding the score, then the initialisation can be part of its constructor:

```
class ScoreCalculator:

    def __init__(self):
        self.score = {'correct position':0, 'wrong position':0}
    # ...
```

Human move

Because the format of the score will differ, the score cannot be used as such. Therefore the while-loop must be changed. To do this, we add a method that returns whether the guess is correct:

```
while not score_calculator.right_guess():
    guess = [int(c) for c in input("Your next guess: ") ]
    score_calculator.score = score_calculator.determine_score(guess, code)
    print('Your score:', score_calculator.score)
```

We also need to be able to reset the score, so the methods are added to do that:

```
class ScoreCalculator:

    def __init__(self):
        self.score = {'correct position':0, 'wrong position':0}

    def right_guess(self):
        return self.score['correct position'] == 4

    def reset_score(self):
        self.score = {'correct position':0, 'wrong position':0}
```

Computer move

The next step is to look at the code for the computer as codebreaker:

```
score = {'correct position':0, 'wrong position':0}
while score['correct position'] != 4:
    score = do_computer_move(combinations, difficulty_level)
```

This changes to:

```
score_calculator.reset_score()
while not score_calculator.right_guess():
    do_computer_move(score_calculator, combinations, difficulty_level)
```

This shows that, to prepare for the different scoring method, we need to add a method to the **`ScoreCalculator`** class to input the score as well:

```
class ScoreCalculator:
    # ...
    def input_score(self):
        self.score['correct position'] = int(input('How many in the correct position? '))
        if not self.right_guess():
            self.score['wrong position'] = int(input('How many in the wrong position? '))
```

This must also be used in the **`do_computer_move()`** function:

```
def do_computer_move(score_calculator, combinations_left, difficulty_level):
    # ...
    score_calculator.input_score()
    score_calculator.process_score(combinations_left, guess)
```

Score calculators module

To make our program more manageable, we're going to create a Python module "score_calulators" that will hold the **`ScoreCalculator`** class and its subclasses.

Using the following statement, all the classes are imported into the namespace of the main-program:

```
from score_calculators import *
```

Note: be sure to add the path to the source directory.

New score calculator

After having done all the groundwork, the functionality for a new scoring method can be added fairly simply. The first step is creating a subclass of the **ScoreCalculator**:

```
class DifferenceScoreCalculator(ScoreCalculator):
    def reset_score(self):
        self.score = 99

    def input_score(self):
        self.score = int(input('What\'s my score? '))

    def determine_score(self, guess, code):
        self.score = sum(abs(guess[i] - code[i]) for i in range(len(guess)))
        return self.score

    def is_code_correct(self):
        return self.score == 0
```

With the **set_score_calculator()** function instantiating the correct score calculator:

```
def set_score_calculator():
    if input("Specify the scoring type [p=position, d=difference]: ") == 'p':
        return ScoreCalculator()
    else:
        return DifferenceScoreCalculator()
```

We can to let the player input the scoring type:

```
score_calculator = set_score_calculator()
```

Computer result

At the higher levels, the new scoring type quickly becomes very challenging for humans. However, no problem for the computer:

```
My guess: (1, 0, 3, 8)
What's my score? 18
My guess: (7, 2, 1, 0)
What's my score? 12
My guess: (8, 3, 7, 4)
What's my score? 18
My guess: (3, 7, 0, 2)
What's my score? 2
My guess: (3, 8, 1, 2)
What's my score? 0
YES!
```

References

1. Python website
2. PyDev; Python IDE for Eclipse
3. Digitmind code on GitHub
4. IEEE Spectrum programming language ranking
5. TensorFlow