Text puzzle generator documentation - version 1.1

Krzysztof Banecki

January 2021

Contents

Introduction

Text puzzle generator is a program for generating text puzzles into `.pdf` file. Program input consist of the dimensions of the puzzle grid and a random seed. The output is a `.pdf` file with a generated puzzle. The program can has its own Graphical User Interface(GUI) which can be run directly on the Unix devices and an `.exe` file to run on Windows.

In the first chapter general ideas and mechanics of the program are given. Second chapter consists of the detailed description of all the functions written and used by the program. These functions are stored in three separate files: `puzzle_class.py`, `generating_categories_functions.py` and `pdf_printing_functions.py`. Additional diagnostic functions are stored in `statistics_functions.py`. In the third chapter there is a detailed information about the drawing schemes and probabilities used in the program as well as a basic statistics on the operation of the program. In the Chapter 4 information about the GUI is provided as well as the instructions on how to compile the file to the Windows executable format. GUI to run on the Unix machines can be started from `app.py` file. In the last chapter basic examples are presented.

1 General Mechanics

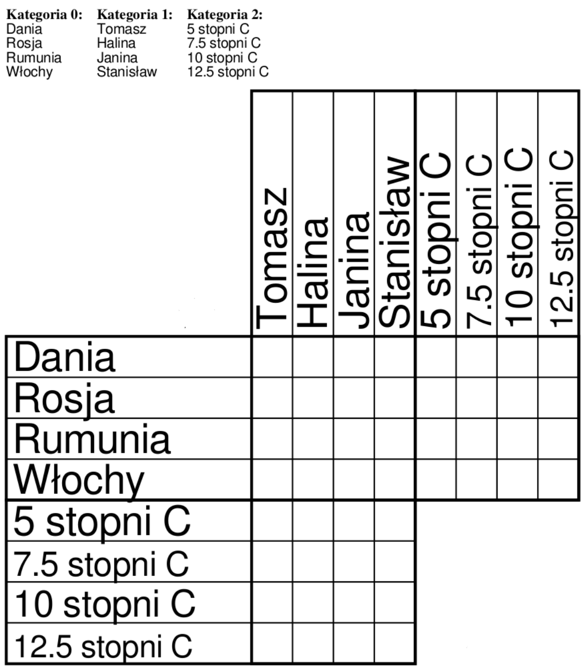
1.1 Puzzle rules

Handling of the text puzzle generation is object oriented. Information about every puzzle is contained within the object of class `puzzle`. Detailed description of the class `puzzle`, its properties and methods are all provided in Section 2.1. Puzzle contains of `K>3` sets of objects (categories) each consisting of `k>3` objects. The goal for the person who solves the puzzle is to bijections between all the sets of the puzzle. That is, every object from the i-th set has to match exactly one object from the first set, exactly one object from the second and from every other set. Moreover no two objects from a i-th set can match the same object from other sets. Every puzzle has its own set of clues that has to be used in order to find the solution which is the total set of matches between puzzle objects. Number of clues is independent from the puzzle dimensions. Clues must unambiguously define the solution. There are currently 6 types of clues implemented and are described in detail in subsections 2.1.15 and 2.1.20.

Human solver should use puzzle grid to mark every match they are sure of with sign "●" and every match that is impossible with sign "$\times$". Grid is initially empty and consists of $K \cdot (K-1)/2$ boxes (one box for every category pair) each of k rows and k columns. Example empty grid is shown at the Figure 1.

After the puzzle is solved all the squares in all of the grid boxes should be filled with signs "●" and sign "$\times$" so that in every row and column of a box there will be exactly one match ("●") and $k-1$ non-matches ("$\times$").

Example clues are presented at the Figure 2 and example solution is depicted at the Figure 3.



Kategoria 0:	Kategoria 1:	Kategoria 2:
Dania	Tomasz	5 stopni C
Rosja	Halina	7.5 stopni C
Rumunia	Janina	10 stopni C
Włochy	Stanisław	12.5 stopni C

Figure 1: Sample initial puzzle grid for $K = 3$ categories with $k = 4$ objects in each.

1.2 Puzzle generator mechanics

As mentioned in the previous section, information about every puzzle is contained within the object of class `puzzle`. Detailed description of the class `puzzle`, its properties and methods are all provided in Section 2.1.

The simplest generating pipeline would be as follows:

- Call the constructor of the function for a given parameters `K>3` and `k>3` in order to obtain an object of class `puzzle`.
- Call the `generate()` method with a given random seed (integer number) in order to generate solvable puzzle.
- Call the `draw_on_canvas` function from the `pdf_printing_functions.py` file on the created object and a given canvas on which it has to be printed.

The example with specific code is provided in Chapter 5.

The crucial role is therefore played by the collective method `generate()`. Function firstly set the random state by the given random seed. It then draw the categories (sets of objects). Since the objects name in most cases have a little impact on how the puzzle is being solved and generated the way program draws those categories is in detail described in Chapter 3 (see also subsection 2.1.36).

1. **Dania** nie pasuje do **Janina**
2. Albo **Tomasz** pasuje do **12.5 stopni C** albo **Halina** pasuje do **10 stopni C** (alternatywa)
3. Pod względem **Kategorii 2** obiekt **Rosja** jest tuż obok **Rumunia**
4. Albo **Rosja** pasuje do **7.5 stopni C** albo **Wlochy** pasuje do **10 stopni C** (alternatywa)
5. Pod względem **Kategorii 2** obiekt **Wlochy** jest tuż obok **Rumunia**
6. Jeśli **Rumunia** pasuje do **Stanislaw**, to **Stanislaw** pasuje do **7.5 stopni C**
   W przeciwnym przypadku **Rumunia** pasuje do **10 stopni C**
7. Albo **Rosja** pasuje do **Tomasz** albo **Tomasz** pasuje do **10 stopni C** (alternatywa)

Figure 2: Sample clues for the puzzle.

Solution:
Dania ~ Stanisław ~ 5 stopni C
Rosja ~ Tomasz ~ 7.5 stopni C
Rumunia ~ Halina ~ 10 stopni C
Włochy ~ Janina ~ 12.5 stopni C

Figure 3: Sample solution to the puzzle with for $K = 3$ categories with $k = 4$ objects in each.

Having categories drawn, function tries to fit the list of clues so that they will not contradict each other and define an unique solution to the puzzle. Clue drawing is done one by one. After adding any clue to the list function checks if the puzzle is currently contradictory and if it is solvable by the program ('solvable by the program means also that the puzzle have a unique solution confirmed by the program). If the puzzle is solvable by the program and not contradictory then no more clues are added to the clue list. If the puzzle is contradictory clues are drawn from the beginning till success (see subsection 2.1.41). Process up to this point takes around 5% of the total generation time. When the clues are finally drawn, the puzzle is technically generated with success. However, at this stage number of clues drawn is usually huge (> 15) and the puzzle is relatively easy. Function performs therefore attempts to throw out as many clues as possible in a way that maintains solvability of the puzzle. This stage of the generation process takes around 90% of the total generation time. Final step is to assess the difficulty of the generated puzzle (with their clues list being restricted). Program solves the final puzzle 5 times (with different random state each time). On each turn it asses the difficulty by counting the number of times the program needed "guessing" in order to find a unique solution. The final difficulty is defined as a mean of difficulties derived from those 5 turns. This stage of the generation process takes around 5% of the total generation time.

As one can see, the generation was using the solving function multiple times in order to generate the final puzzle. This solving task is done by the `try_to_solve()` method of class `puzzle` (see Subsection 2.1.39). This function is a solving function for cracking all the puzzles (including hard ones). Simple solving is implemented in method `try_to_solve2()`. Simple solving runs through all the clues and firstly it tried to use use as much as it can and then concile the grid. By word 'concile' we mean get as much information, that is put as many "●" and "$\times$" signs in to the grid, without looking at any of the clues. This task is done by the method `grid_concile()` (see Subsection 2.1.12). Detailed information about how the grid conciliation is being made is provided in subsections from 2.1.7 to 2.1.12.

Method `try_to_solve()` additionally performs "guessing". That is, for every empty square in the grid it tries to put matching sign there and see if it would be able to solve it now by the method `try_to_solve2()`. If at the end the puzzle ends contradictory `try_to_solve()` can exclude that square from the match possibility (and put "×" sign there). Grid boxes are checked in a random order so the the number of solving steps can differ which is used in generation to assess the difficulty of the puzzle.

2 Functions Descriptions

2.1 puzzle_class.py

Python packages required: `numpy, pandas, copy`.

Uses functions from `generating_categories_functions.py` (see Section 2.2).

File `puzzle_class.py` contains the definition of class `puzzle` and all of its methods described in detail in the following subsections. Properties of class `puzzle` are:

- `K` - number of categories.
- `k` - number of objects in each category.
- `grid` - dictionary of `Numpy` arrays containing current solution state grid. Dictionary is indexed by the strings consisting of the categories indexes separated by a comma (e.g. `self.grid['1,2']` represents an array between second and third category). First number in those indexes are always smaller than the second one. Category with smaller index is aligned as rows in the array whereas category with larger index as columns. That is, the box in row `2` and column `0` shows whether the third object of the category with smaller index is compatible to the first object of the category with larger index (see example below). Information in the `grid` is stored in three states: `0` - when no information about the compatibility is known, `1` - if objects are compatible and `2` - if object are not compatible.
- `changed` - auxiliary flag indicating whether the solution grid has been changed.
- `solved` - auxiliary flag indicating whether the solution grid is solved already.
- `contradictory` - auxiliary flag indicating whether the solution grid is contradictory.

- `categories` - list of categories of size `K` with names of all objects in each category. Every category is represented by a tuple. First element of a tuple describes the type of the category, that is `'categorical'`, `'ordinal'` or `'numerical'`. Second element is a list of object names (or just numbers in the case of numerical category). Objects in numerical and ordinal categories are always aligned from the smallest to the greatest. Numerical categories have also additional third and fourth element. Third one is a set of possible numerical clues and the fourth describes how objects of the category should be printed in the final puzzle view. For example categories set see example below.
- `clues` - list of puzzle clues. Every clue is a dictionary. Under the key `'typ'` there is the type of the clue. Keys `K1-K6` codes the category indexes involved in a clue, keys `i1-i6` codes object indexes in corresponding category and keys `'diff'` and `'oper'` code type of numerical clue. More about clues in Subsection 2.1.37. Example clues are listed in the example below.
- `seed` - random seed.
- `diff` - assessed difficulty of the puzzle as a mean of 5 difficulty assessments.

Example. Let us initialize puzzle for `K=4` and `k=3`. We then set a random seed and generate categories and clues:

```
puzzle1 = puzzle(K=4, k=3)
seed = random.randint(100,100000)
puzzle1.generate(seed)
```

`puzzle1.categories` are:

```
[('categorical', ['bark', 'noga', 'palec']),
 ('ordinal', ['A3', 'A2', 'A1']),
 ('numerical', [2000.0, 2025.0, 2050.0], {'x=y+25'}, '@_gwiazd'),
 ('numerical', [10.0, 20.0, 30.0], {'x=y+10.0'}, '@_EUR')]
```

`puzzle1.grid` look as follows:

```
{'0,1': array([[2., 2., 1.],
               [1., 2., 2.],
               [2., 1., 2.]]),
 '0,2': array([[2., 1., 2.],
               [2., 2., 1.],
               [1., 2., 2.]]),
 '0,3': array([[2., 2., 1.],
               [1., 2., 2.],
               [2., 1., 2.]]),
 '1,2': array([[2., 2., 1.],
               [1., 2., 2.],
               [2., 1., 2.]]),
 '1,3': array([[1., 2., 2.],
               [2., 1., 2.],
               [2., 2., 1.]]),
 '2,3': array([[2., 1., 2.],
               [2., 2., 1.],
               [1., 2., 2.]])}
```

`puzzle1.clues` are:

```
[{'typ':5,'K1':0,'i1':2,'K2':1,'i2':2,
          'K3':0,'i3':1,'K4':1,'i4':1},
 {'typ':3,'K1':1,'i1':2,'K2':1,'i2':1,
          'K6':3,'diff':10.0,'oper':'+'},
 {'typ':3,'K1':0,'i1':1,'K2':0,'i2':2,
          'K6':2,'diff':25.0,'oper':'+'},
 {'typ':4,'K1':2,'i1':0,'K2':3,'i2':0,
        'K3':1,'i3': 1,'K4':2,'i4':0,'K5':1,'i5':2,'K6':3,'i6':0},
 {'typ':4,'K1':1,'i1':1,'K2':2,'i2':0,
        'K3':0,'i3': 2,'K4':2,'i4':2,'K5':0,'i5':0,'K6':2,'i6':1}]
```

2.1.1 grid_insert

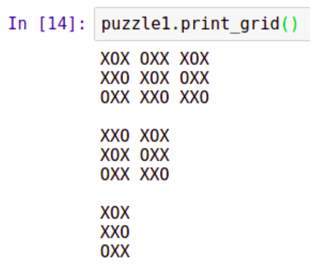
`grid_insert(self, K1, i1, K2, i2, val)` is a method of class `puzzle`. It is a recommended function to make changes into the `grid`. It inserts a new value to the `grid` regardless of the previous value of the considered square. If the previous value was different than the new one function sets `self.changed` flag as `True`.

Value `val` can be either one of `0, 1, 2` or one of `'-', 'O', 'X'` respectively. It is recommended to use this function with string notation.

The value `val` is inserted in the square of correspondence between `i1`-th object of category `K1` and `i2`-th object of category `K2`. Index `K1` do not have to be smaller than index `K2`. Function takes care of the proper insert into the `grid` dictionary.

2.1.2 print_grid

`print_grid(self)` is a method of class `puzzle`. Function prints the current state of the `grid` in the graphical form (with `'-', 'O', 'X'` signs) to the terminal.



```
In [14]: puzzle1.print_grid()

XOX OXX XOX
XXO XOX OXX
OXX XXO XXO

XXO XOX
XOX OXX
OXX XXO

XOX
XXO
OXX
```

Figure 4: Example `print_grid` output of a fully solved puzzle.

The first row of boxes is assigned to the category `0`. Second to the category `1` and so on. Finally, the last row to the category `K-2`. Columns are numbered from the category `K-1` on the left column to the category `1` on the right.

2.1.3 clear_grid

`clear_grid(self)` is a method of class `puzzle`. Function sets all of the values in the `grid` to `0`.

2.1.4 get_grid_value

`get_grid_value(self, K1, i1, K2, i2)` is a method of class `puzzle`. It is a recommended function to get `grid` values. Function returns value (`0, 1, 2`) of the correspondence between `i1`-th object of category `K1` and `i2`-th object of category `K2`. Index `K1` do not have to be smaller than index `K2`.

2.1.5 is_line_completed

`is_line_completed(self, K1, i1, K2)` is an auxiliary method of class `puzzle`. It checks if the `i1`-th object from category `K1` has exactly one match in the current `grid` (`'O'`) with regards to category `K2` and all the other possibilities are excluded (`'X'`). Returns a boolean value.

2.1.6 count_x_in_line

`count_x_in_line(self, K1, i1, K2)` is an auxiliary method of class `puzzle`. It checks how many possibilities are excluded (`'X'`) for the `i1`-th object from category `K1` with regards to category `K2`. Returns an integer value.

2.1.7 grid_concile1

`grid_concile1(self)` is a method of class `puzzle`. Performs the first type of grid conciliation based solely on the current grid values (regardless of the puzzle clues). If there is an `'O'` in the square anywhere in the `grid` function adds `'X'`s in all directions (cross like). See Figure 5.

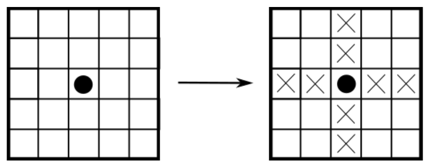


Figure 5: First type of grid conciliation.

2.1.8 grid_concile2

`grid_concile2(self)` is a method of class `puzzle`. Performs the second type of grid conciliation based solely on the current grid values (regardless of the puzzle clues). If there is only one blank square(`'_'`) in row/column function inserts `'O'` in it (and add `'X'`'s in all directions from it). See Figure 6.

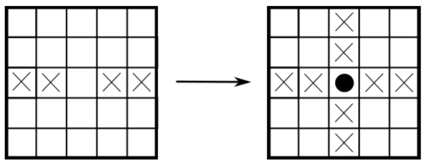


Figure 6: Second type of grid conciliation.

2.1.9 grid_concile3

`grid_concile3(self)` is a method of class `puzzle`. Performs the third type of grid conciliation based solely on the current grid values (regardless of the puzzle clues). If there is an `'O'` in the box linking object `i1` to object `i2`, function conciles their `'X'`'s. That is, if object `i1` has an `'X'` with some other object, `i2` has to had `'X'` with it too. See Figure 7.

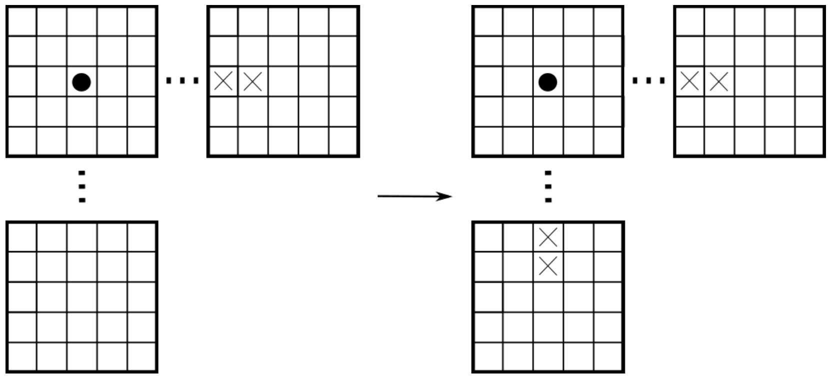


Figure 7: Third type of grid conciliation.

2.1.10 grid_concile4

`grid_concile4(self)` is a method of class `puzzle`. Performs the fourth type of grid conciliation based solely on the current grid values (regardless of the puzzle clues). Function looks for contradictory lines. If two lines with respect to some category `K1` contradicts each other (that is, in summary they exclude all of the options), then the objects associated with that lines cannot be linked together (therefore we put `'X'` there). See Figure 8.

2.1.11 grid_concile5

`grid_concile5(self)` is a method of class `puzzle`. Performs the fifth type of grid conciliation based solely on the current grid values (regardless of the puzzle clues). If two/three line's blank squares are restricted only to two/three the

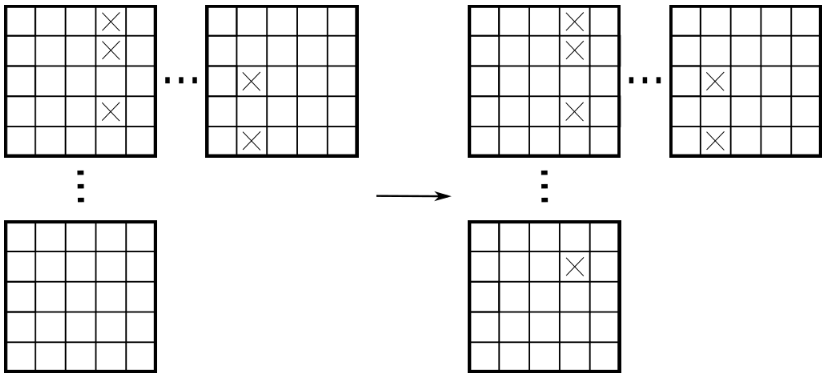


Figure 8: Fourth type of grid conciliation.

same options, then no other line can have `'O'` on the level of that blank square. See Figure 9.

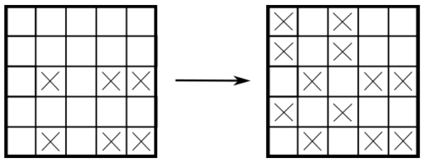


Figure 9: Fifth type of grid conciliation.

2.1.12 grid_concile

`grid_concile(self)` is a collective method of class `puzzle`. It is a recommended function to reconcile the `grid`. Function calls all described conciliation methods from `grid_concile1` to `grid_concile5` until no more progress can be achieved.

Function is using `self.changed` flag to store the information whether the change has been made. Currently the final state of this flag after the use of this function is always `False`. Version in which function maintains the current flag status if no change was made within it and change it to `True` otherwise is currently commented in the code due to numerical tediousness it produces.

2.1.13 get_critical_squares

`get_critical_squares(self, clue)` is an auxiliary method of class `puzzle`. It is used by the function `is_forbidden()` (see Subsection 2.1.14) in order to determine if clue candidate can be added to the puzzle without too easily resolve any other clue. Function returns the critical squares for the clue (or candidate for a clue) provided. Critical squares are those on which solver would immediately put `'X'` signs on or that on which every change would in significant way affect the clue. Critical squares depending on the type of the clue are:

- For the clue of type 1: square `(K1,i1,K2,i2)`, and squares `(K1,i1,K2,i3)` and `(K1,i1,K2,i4)` if `i3` and `i4` are present in the clue.
- For the clue of type 2:
  - squares `(K1,i1,K6,0)` and `(K1,i1,K6,1)`,
  - squares `(K2,i2,K6,0)` and `(K2,i2,K6,k-1)`,
  - squares `(K3,i3,K6,k-1)` and `(K3,i3,K6,k-2)`,
  - if objects involved are in different categories squares at their intersections.
- For the clue of type 3:
  - if objects involved are in different categories squares at their intersections.
  - all squares that do not have the square with respective value resulting from the clue relationship.
- For the clue of type 4:
  - squares `(K1,i1,K2,2)` and `(K3,i3,K4,i4)`.
  - if squares `(K1,i1,K2,2)` and `(K3,i3,K4,i4)` are in the same box then all squares on which by putting the match sign there both of the above squares would immediately be excluded.
- For the clue of type 5:
  - squares `(K1,i1,K2,2)`, `(K3,i3,K4,i4)` and `(K5,i5,K6,i6)`.
  - if squares `(K3,i3,K4,i4)` and `(K5,i5,K6,i6)` are in the same box then all squares on which by putting the match sign there both of the above squares would immediately be excluded.
- For the clue of type 6: If objects are from different category, square `(K1,i1,K2,2)`.

Function returns list of critical squares in the form of `(K1,i1,K2,i2)`.

2.1.14 is_forbidden

`is_forbidden(self, clue_cand)` is an auxiliary method of class `puzzle`. Function is called before every clue is being added to the puzzle, therefore its aim is to prevent clues that directly solve other clues from being attached to the clues list (see Subsection 2.1.37).

Firstly, function uses the function `get_critical_values()` (see Subsection 2.1.13) to check if any of the critical squares of the candidate for a clue `clue_cand` coincides with any of the critical squares of any of the clues in puzzle clue list.

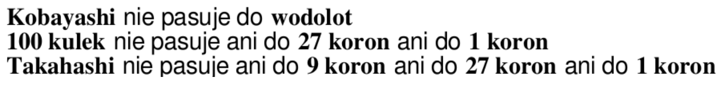
The rest of the function is called only for clues of type 2 or 3. Function prevents from the situation in which there are two clues of type 2 of 3 which directly solve one of the objects in a box (that is situation in which in some line those two clues provide `k-1` `'X'` signs). Firstly it checks if such situation can occur for two clues of type 2. Then is checks the same for the one clue 3 and 2 with the basic assumption of a single `'X'` in the clue of type 3. And finally it uses critical squares for general case.

Returns a boolean value.

2.1.15 add_clue1

`add_clue1(self)` is a method of class `puzzle`. Function tries to add another clue of type `1` to the current clues list (`self.clues`). The example clues of type `1` are:

```
{'typ': 1, 'K1': 2, 'i1': 1, 'K2': 3, 'i2': 1}
{'typ': 1, 'K1': 1, 'i1': 0, 'K2': 3, 'i2': 3, 'i3': 2}
{'typ': 1, 'K1': 0, 'i1': 0, 'K2': 3, 'i2': 2, 'i3': 3, 'i4': 0}
```



Kobayashi nie pasuje do **wodolot**
100 kulek nie pasuje ani do **27 koron** ani do **1 koron**
Takahashi nie pasuje ani do **9 koron** ani do **27 koron** ani do **1 koron**

Figure 10: Graphical view of the example clues of type 1.

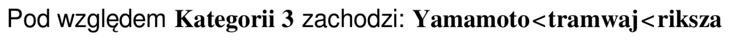
Clues of type `1` have 3 subtypes shown in the above example. First subtype describes a incompatibility between `i1`-th object from category `K1` and `i2`-th object from category `K2`. Second subtype also between `i3`-th object from category `K2`. Third subtype also between `i4`-th object from category `K2`. Probabilities of drawing one of those subtypes are set to: 80%, 15% and 5% respectively.

Function calls `is_forbidden()` function before every clue is added to the clues list. Function will look for the clue which crucial squares were all empty (`'_'`). If function fails to find such squares no clue is added to the clues list.

2.1.16 add_clue2

`add_clue2(self)` is a method of class `puzzle`. Function tries to add another clue of type `2` to the current clues list (`self.clues`). The example clue of type `2` is:

```
{'typ':2,'K1':2,'i1':1,'K2':2,'i2':3,'K3':2,'i3':2,'K6':3}
```



Pod względem **Kategorii 3** zachodzi: **Yamamoto<tramwaj<riksza**

Figure 11: Graphical view of the example clue of type 2.

Clue of this type describes a situation in which with respect to the category `K6` (which must be either numerical or ordinal) we have the following relationship:

`i3`-th object from category `K3` is strictly smaller than `i2`-th object from category `K2` which is strictly smaller than `i1`-th object from category `K1`.

In short: '`K3,i3 < K2,i2 < K1,i1`'.

Function calls `is_forbidden()` function before every clue is added to the clues list. Function will look for the clue which crucial squares were all empty (`'_'`). If function fails to find such squares no clue is added to the clues list.

2.1.17 add_clue3

`add_clue3(self)` is a method of class `puzzle`. Function tries to add another clue of type `3` to the current clues list (`self.clues`). The example clue of type `3` is:

```
{'typ': 3, 'K1': 1, 'i1': 3, 'K2': 0, 'i2': 2, 'K6': 3,
  'diff': 9.0, 'oper': '*'}
```

Pod względem **Kategorii 3** zachodzi: **wodolot = 200 kulek** * 9

Figure 12: Graphical view of the example clue of type 3.

Clue of this type describes a situation in which with respect to the category `K6` (which must be numerical) we have the following relationship:

- if `oper=='+'`:

  the value `i1`-th object from category `K1` plus the value of `diff` equals the value `i2`-th object from category `K2`.

- if `oper=='*'`:

  the value `i1`-th object from category `K1` multiplied by the value of `diff` equals the value `i2`-th object from category `K2`.

In short: '`K2,i2 = K1,i1 oper diff`', where `oper` is one of $\{+, *\}$.

Function calls `is_forbidden()` function before every clue is added to the clues list. Function will look for the clue which crucial squares were all empty (`'_'`). If function fails to find such squares no clue is added to the clues list.

2.1.18 add_clue4

`add_clue4(self)` is a method of class `puzzle`. Function tries to add another clue of type `4` to the current clues list (`self.clues`). The example clue of type `4` is:

```
{'typ': 4, 'K1': 1, 'i1': 2, 'K2': 2, 'i2': 1,
  'K3': 2, 'i3': 3, 'K4': 3, 'i4': 1,
  'K5': 2, 'i5': 1, 'K6': 3, 'i6': 3}
```

Jeśli **Takahashi** pasuje do **autobus**, to **300 kulek** pasuje do **9 koron**
W przeciwnym przypadku **Yamamoto** pasuje do **100 kulek**

Figure 13: Graphical view of the example clue of type 4.

Clue of this type describes the following situation:

If `i1`-th object from category `K1` matches the `i2`-th object from category `K2` then `i3`-th object from category `K3` has to match the `i4`-th object from category `K4`, otherwise `i5`-th object from category `K5` matches the `i6`-th object from category `K6`.

In short: 'If `K1,i1`∼`K2,i2` then `K3,i3`∼`K4,i4` otherwise `K5,i5`∼`K6,i6`'.

There is a 50% chance that categories `K3` and `K4` determine the same grid box as categories `K5` and `K6`. In this case there are more crucial squares for the function to look at.

Function calls `is_forbidden()` function before every clue is added to the clues list. Function will look for the clue which crucial squares were all empty (`'_'`). If function fails to find such squares no clue is added to the clues list.

2.1.19 add_clue5

`add_clue5(self)` is a method of class `puzzle`. Function tries to add another clue of type `5` to the current clues list (`self.clues`). The example clues of type `5` are:

```
{'typ': 5, 'K1': 1, 'i1': 3, 'K2': 3, 'i2': 0,
  'K3': 1, 'i3': 0, 'K4': 3, 'i4': 2},
{'typ': 5, 'K1': 1, 'i1': 2, 'K2': 2, 'i2': 0,
  'K3': 0, 'i3': 0, 'K4': 2, 'i4': 2}
```

Albo **autobus** pasuje do **1 koron** albo **riksza** pasuje do **9 koron** (alternatywa)
Albo **wodolot** pasuje do **100 kulek** albo **Takahashi** pasuje do **300 kulek** (alternatywa)

Figure 14: Graphical view of the example clues of type 5.

Clue of this type describes the following situation:

Either `i1`-th object from category `K1` matches the `i2`-th object from category `K2` or `i3`-th object from category `K3` matches the `i4`-th object from category `K4`.

In short: 'Either `K1,i1`∼`K2,i2` or `K3,i3`∼`K4,i4`'.

There is a 50% chance that categories `K1` and `K2` determine the same grid box as categories `K3` and `K4`. This situation is shown on the first example clue (notice that `K1=K3` and `K2=K4`). Second example shows the opposite situation. In the case of equality there are more crucial squares for the function to look at.

Function calls `is_forbidden()` function before every clue is added to the clues list. Function will look for the clue which crucial squares were all empty (`'_'`). If function fails to find such squares no clue is added to the clues list.

2.1.20 add_clue6

`add_clue6(self)` is a method of class `puzzle`. Function tries to add another clue of type `6` to the current clues list (`self.clues`). The example clues of type `6` are:

```
{'typ': 6, 'K1': 1, 'i1': 1, 'K2': 2, 'i2': 1, 'K6': 3}
{'typ': 6, 'K1': 2, 'i1': 3, 'K2': 2, 'i2': 0, 'K6': 3}
```

Pod względem **Kategorii 3** obiekt **tramwaj** jest tuż obok **200 kulek**
Pod względem **Kategorii 3** obiekt **400 kulek** jest tuż obok **100 kulek**

Figure 15: Graphical view of the example clues of type 6.

Clue of this type describes a situation in which with respect to the category `K6` (which must be either numerical or ordinal) we have the following relationship:

`i1`-th object from category `K1` is right next to the `i2`-th object from category `K2`.

Categories `K1` and `K2` may or may not be equal (in the first example they are not, in the second they are).

Function calls `is_forbidden()` function before every clue is added to the clues list. Function will look for the clue which crucial squares were all empty (`'_'`). If function fails to find such squares no clue is added to the clues list.

2.1.21 use_clue1

`use_clue1(self, c)` is an auxiliary method of class `puzzle`. Function applies the clue of index `c` in the clues list(`self.clues`) to the `grid`. One can use this function only if this particular clue is of type `1`. It is recommended to use collective function `use_clue()` (see Subsection 2.1.27).

Function puts `'X'` into the grid in the square determined by `(K1, i1, K2, i2)` as well as in the squares `(K1, i1, K2, i3)` and `(K1, i1, K2, i4)` if present in within the clue (see Subsection 2.1.15 for example).

2.1.22 use_clue2

`use_clue2(self, c)` is an auxiliary method of class `puzzle`. Function applies the clue of index `c` in the clues list(`self.clues`) to the `grid`. One can use this function only if this particular clue is of type `2`. It is recommended to use collective function `use_clue()` (see Subsection 2.1.27).

Function firstly puts `'X'` at two squares with lowest value with respect to category `K6` for the object `K1,i1`, the highest and lowest value for object `K2,i2` and at two squares with highest value for the object `K3,i3`.

Then if the objects belong to different categories they cannot match with each other so the function excludes that possibility.

Finally function excludes squares with too large value with respect to category `K6` for the objects that are suppose to be smaller and squares with too large values for the objects that are suppose to be larger.

2.1.23 use_clue3

`use_clue3(self, c)` is an auxiliary method of class `puzzle`. Function applies the clue of index `c` in the clues list(`self.clues`) to the `grid`. One can use this function only if this particular clue is of type `3`. It is recommended to use collective function `use_clue()` (see Subsection 2.1.27).

Firstly if the objects belong to different categories they cannot match with each other so the function excludes that possibility.

Then it excludes squares for which the square of the other object with matching clue relationship has been already eliminated (`'X'`).

2.1.24 use_clue4

`use_clue4(self, c)` is an auxiliary method of class `puzzle`. Function applies the clue of index `c` in the clues list(`self.clues`) to the `grid`. One can use this function only if this particular clue is of type `4`. It is recommended to use collective function `use_clue()` (see Subsection 2.1.27).

Firstly if categories `K3` and `K4` determine the same grid box as categories `K5` and `K6` function rules out all the squares in this box that will immediately exclude both of squares `(K3,i3,K4,i4)` and `(K5,i5,K6,i6)`.

Then it applies the following rules:

- If square `(K3,i3,K4,i4)` has been already excluded(`'X'`) then exclude square `(K1,i1,K2,i2)`.
- If square `(K5,i5,K6,i6)` has been already excluded(`'X'`) then put `'O'` in the square `(K1,i1,K2,i2)`.
- If square `(K1,i1,K2,i2)` contains `'O'` in it then put `'O'` in the square `(K3,i3,K4,i4)`.
- If square `(K1,i1,K2,i2)` contains `'X'` in it then put `'O'` in the square `(K5,i5,K6,i6)`.

2.1.25 use_clue5

`use_clue5(self, c)` is an auxiliary method of class `puzzle`. Function applies the clue of index `c` in the clues list(`self.clues`) to the `grid`. One can use this function only if this particular clue is of type `5`. It is recommended to use collective function `use_clue()` (see Subsection 2.1.27).

Firstly if categories `K1` and `K2` determine the same grid box as categories `K3` and `K4` function rules out all the squares in this box that will immediately exclude both of squares `(K1,i1,K2,i2)` and `(K3,i3,K4,i4)`.

Then it applies the following rules:

- If square (K1,i1,K2,i2) contains 'X' in it then put 'O' in the square (K3,i3,K4,i4).
- If square (K3,i3,K4,i4) contains 'X' in it then put 'O' in the square (K1,i1,K2,i2).

2.1.26 use_clue6

use_clue6(self, c) is an auxiliary method of class puzzle. Function applies the clue of index c in the clues list(self.clues) to the grid. One can use this function only if this particular clue is of type 6. It is recommended to use collective function use_clue() (see Subsection 2.1.27).

Function for every empty square of one of the objects checks if it has at least one empty square of the other object nearby and puts exclude that square otherwise.

2.1.27 use_clue

use_clue(self, c) is an collective method of class puzzle. It is a recommended function to apply any clue to a current solution grid. It redirects to one the functions use_clue1 to use_clue6 with regard to the given clue index c.

2.1.28 is_grid_contradictory_with_clue2

is_grid_contradictory_with_clue2(self, c) is an auxiliary method of class puzzle. Function checks if there exist a way of aligning the objects involved in the clue in the given relationship (see Subsection 2.1.16). Returns boolean value.

2.1.29 is_grid_contradictory_with_clue3

is_grid_contradictory_with_clue3(self, c) is an auxiliary method of class puzzle. Function checks if there exist a way of aligning the objects involved in the clue in the given relationship. (see Subsection 2.1.17). Returns boolean value.

2.1.30 is_grid_contradictory_with_clue4

is_grid_contradictory_with_clue4(self, c) is an auxiliary method of class puzzle. Function checks for the occurrence of the following situations:

- if both square (K3,i3,K4,i4) and (K5,i5,K6,i6) have been already excluded then return True.
- if square (K1,i1,K2,i2) is a match and (K3,i3,K4,i4) has been already excluded then return True.
- if both square (K1,i1,K2,i2) and (K5,i5,K6,i6) have been already excluded then return True.

Returns boolean value.

2.1.31 is_grid_contradictory_with_clue5

`is_grid_contradictory_with_clue5(self, c)` is an auxiliary method of class `puzzle`. Function returns `True` if both squares `(K1,i1,K2,i2)` and `(K3,i3,K4,i4)` have been already excluded and `False` otherwise. Returns boolean value.

2.1.32 is_grid_contradictory_with_clue6

`is_grid_contradictory_with_clue6(self, c)` is an auxiliary method of class `puzzle`. Function firstly looks for a number of possible pairs of values that can be assign to the objects involved in the clue and returns `True` if there are no such pairs. Then function checks for matches for considered objects and checks if they have possible non-excluded squares of the other object. Returns boolean value.

2.1.33 is_grid_completed

`is_grid_completed(self)` is an auxiliary method of class `puzzle`. It checks if there is at least one square in the whole `grid` with `'-'` mark. Returns boolean value.

2.1.34 is_grid_contradictory

`is_grid_contradictory(self)` is a method of class `puzzle`. It checks all the rows and columns of every box in the `grid` in search of more than a one `'O'`'s or entirely filled with `'X'`'s. In that case function returns `True`. Function then for every clue calls an appropriate function from `is_grid_contradictory_with_clue2()` to `is_grid_contradictory_with_clue6()`. Returns boolean value.

2.1.35 set_seed

`set_seed(self, seed)` is a method of class `puzzle`. Function sets random seed `seed`. This function is used directly at the beginning of the function `self.generate` (see Subsection 2.1.41).

2.1.36 draw_categories

`draw_categories(self, diff=3)` is a method of class `puzzle`. Function draws the categories for the puzzle. It calls an appropriate function from the `generating-_categories_functions.py` file (see Subsection 2.2.5) until none of the objects drawn repeat.

Parameter `diff` is transferred to the function in other file and is responsible for different probabilities of category classes that are: *categorical, numerical* and *ordinal.*

2.1.37 draw_clues

`draw_clues(self, trace=False)` is a method of class `puzzle`. Function draws the clues for the puzzle. Function assumes categories have been already drawn before. Function firstly tries to draw clues of all the types other than `1`. Number of such clues is determine by the equation:

$$\text{Number of non-1 clues } = \lceil k \cdot K/2.3 \rceil. \tag{1}$$

For example this number is equal to 9 for `k=4` and `K=5` and equal 4 for `k=3` and `K=3`. This parameter is larger for larger values of `k` and `K` (larger grids) and smaller for smaller values (smaller grids).

Probabilities of drawing certain type of clue are set to 20% for every clue type: 2, 3, 4, 5 and 6. Function updates the grid after adding any new clue both by using clue and conciling the grid. If the puzzle can be solved before drawing all the clues function finish. If puzzle has not been solved at this point function tries to fit as many (with the limit of 100) clues of type `1` (with grid updating in between) to make puzzle solvable or contradictory. Function stop if the puzzle with resulting clue list is either solvable or contradictory.

Additionally function has its own control parameter `trace`. If `trace=True` diagnostic information will be printed to the console.

2.1.38 try_to_solve2

`try_to_solve2(self)` is an auxiliary method of class `puzzle`. Function tries to solve the puzzle by alternately using all the clues (see Subsection 2.1.27) and conciling the grid (see Subsection 2.1.12) until no change is made. Function is using `self.changed` flag to store the information whether the change has been made. Currently the final state of this flag after the use of this function is always `False`. Version in which function maintains the current flag status if no change was made within it and change it to `True` otherwise is currently commented in the code due to numerical tediousness it produces.

2.1.39 try_to_solve

`try_to_solve(self, max_iter=None)` is a method of class `puzzle`. This is a recommended function to use in order to try to solve the puzzle. Function for every square in the grid tries to put `'O'` mark and check if it can be solved now by `try_to_solve2()`. In case the puzzle is confirmed contradictory after this operation function puts `'X'` in this square and try to solve the puzzle again. If puzzle is still contradictory function sets the flag `self.contradictory` as `True` and finish. Otherwise it continues till solves the puzzle, in which case it sets the flag `self.solved` as `True` and finish, or it runs out of squares.

If `max_iter` is specified (as a positive integer value) then function will check only up to `max_iter` random squares to put `'O'` in it and try to solve. This option can be used in order to significantly reduce computational tediousness in reducing clues number phase (see Subsection 2.1.40).

2.1.40 try_to_restrict_clues

`try_to_restrict_clues(self, trace=False, max_iter=None)` is a method of class `puzzle`. Function tries to remove as many clues from the clues list so that the resulting puzzle could stay solvable. Function starts with the clues of the type `1` and then proceed to the other types of clues in random order. Function clears the grid, erase one considered clue from the clue list and try to solve the puzzle using `try_to_solve()` function (see Subsection 2.1.39). If this function were able to solve the puzzle even without this clue it is removed and next clues are tested the same way under the reduced clue list.

If `max_iter` is specified it is transferred to the function `try_to_solve()` with every function call. If `trace=True` diagnostic information will be printed to the console.

2.1.41 generate

`generate(self, seed=0, trace=False, max_iter=None)` is a method of class `puzzle`. It is a collective function that generates the whole puzzle from scratch. The steps performed by the function are:

- Set the random seed as `seed` using function described in Subsection 2.1.35.
- Draw categories using function described in Subsection 2.1.36.
- Draw clues using function described in Subsection 2.1.37. Since this function may fail to draw those clues, function tries to draw them 100 times till success.
- Reduce number of clues using function described in Subsection 2.1.40. Parameters `trace` and `max_iter` are passed to that function.
- Finally, assign the difficulty to the resulting puzzle. Final difficulty is defined as the mean of 5 independent solving approaches by the function `try_to_solve()` and rounding the result to one decimal place. In this step parameters `trace` and `max_iter` are not passed to the auxiliary function.

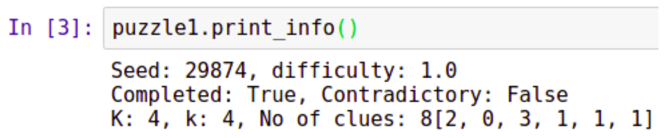
2.1.42 print_info

`print_info(self)` is an auxiliary method of class `puzzle`. It print to the console basic information about the puzzle like: seed, difficulty, information whether the puzzle is solved and whether is contradictory, parameters `self.K` and `self.k`, number of clues and counts of every clue type.

2.2 generating_categories_functions.py

Python packages required: `numpy, pandas, pathlib`.

In this file all the functions for the categories generation are stored. These functions are used by the `draw_categories` method of class `puzzle`.



```
In [3]: puzzle1.print_info()

Seed: 29874, difficulty: 1.0
Completed: True, Contradictory: False
K: 4, k: 4, No of clues: 8[2, 0, 3, 1, 1, 1]
```

Figure 16: Example output of the function `print_info()`.

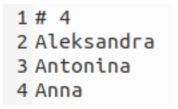
2.2.1 draw_cat_scheme

`draw_cat_scheme(ile=1, puzzle_k=5)` is an auxiliary external function. Function draws `ile` sets of objects from a categorical category, `puzzle_k` objects in each set. First, categories are chosen for each set of objects. All categorical categories are stored in the `./categories/categorical/` folder. Those categories are then divided into 7 separate classes/folders, each of those classes containing some number of categories, each of category containing some number of objects. Each class of categories has its own probability weight assigned as it is shown at the below table:

Category class	Weight
activities	3
geography	3
items	5
names	12
nature	4
other_concepts	5
special_names	6

If `ile` is small (`<6`) then there is a 75% chance of drawing without replacement and 25% of drawing with replacement. Otherwise function always draws with replacement. More details can be found in Section 3.

After folders for each set of objects have already been selected function in each folder filter the categories that has too little objects in it to draw from (`<puzzle_k`). Then draw the categories for each set with regards to the probability weights specified in `.txt` files containing certain category.



```
# 4
Aleksandra
Antonina
Anna
```

Figure 17: Example beginning of the `.txt` file of a certain categorical category.

There are two special categories which has to be treated differently. These are:

- `special_names/zmysleni_superbohaterowie_ang.txt`
- `special_names/zmysleni_superbohaterowie_pol.txt`

In their case function additionally draws both nickname and name of an object, trying to balance the male and the female objects.

Function returns a list of categories like the following:

```
[('categorical', ['monocykl', 'szosowy', 'mtb']),
 ('categorical', ['owady', 'skorupiaki', 'ssaki']),
 ('categorical', ['Pimpek', 'Iwo', 'Zigzak'])]
```

2.2.2 draw_ord_scheme

`draw_ord_scheme(ile=1, puzzle_k=5)` is an auxiliary external function. Function draws `ile` sets of objects from a ordinal category, `puzzle_k` objects in each set. First, categories are chosen for each set of objects. All ordinal categories are stored in the `./categories/ordinal/` folder. Object from the ordinal kind of category are assumed to have a well defined ordinal relation (`<`) and that they are aligned in the ascending order according to this relation.

Function filter the categories that has too little objects in it to draw from (`<puzzle_k`). Then draw the categories for each set with regards to the probability weights specified in `.txt` files containing certain category the same way it was done in the previous subsection. Finally, function chooses `puzzle_k` not by uniform draw but by taking random `puzzle_k` consecutive objects. Returns a list of categories like the following:

```
[('ordinal', ['milimetr', 'centymetr', 'decymetr']),
 ('ordinal', ['L', 'XL', 'XXL']),
 ('ordinal', ['bardzo_zimno', 'zimno', 'letnio'])]
```

2.2.3 draw_num_scheme

`draw_num_scheme(puzzle_k=5)` is an auxiliary external function. Function draws `ile` numerical values and proposed candidates for the clues of type 3 for just one single numerical category. Interpretation for the numerical value is drawn later in a separate function described in Subsection 2.2.4.

Firstly function draws a set of objects - `puzzle_k` numerical values. Numerical values come in three types(schemes): *"arithmetic sequence"*, *"geometric sequence"* and *"ascending sequence"*. Probability weights and examples of those types of sequences are shown in the table below:

Numerical scheme	Weight	Example (for `puzzle_k=6`)
arithmetic sequence	10	$1, 2, 3, 4, 5, 6$
geometric sequence	2	$1, 2, 4, 8, 16, 32$
ascending sequence	4	$1, 2, 4, 5, 7, 8$

Regardless of the numerical scheme, specific number are drawn basing on three factors: base value `p`, times factor `n` and increment/multiplier `r`.

- If the arithmetic sequence is chosen final numerical values will be:

$$\mathtt{p} \cdot \mathtt{n}, (\mathtt{p} \cdot \mathtt{n}) + \mathtt{r}, (\mathtt{p} \cdot \mathtt{n}) + 2\mathtt{r}, (\mathtt{p} \cdot \mathtt{n}) + 3\mathtt{r}, \ldots$$

- If the geometric sequence is chosen final numerical values will be:

$$\mathtt{p}\cdot\mathtt{n}, (\mathtt{p}\cdot\mathtt{n})\cdot\mathtt{r}, (\mathtt{p}\cdot\mathtt{n})\cdot\mathtt{r}^2, (\mathtt{p}\cdot\mathtt{n})\cdot\mathtt{r}^3, \ldots$$

- If the ascending sequence is chosen final numerical values will look similar to the arithmetic sequence but from time to time `2r` will be added instead of `r`. Additionally, increment `r` is always equal to `n`·`p` in this case.

Firstly base value `p` is drawn from the given set with its probabilities. Depending on the numerical schemes some of the base values are excluded. Then times factor `n` is drawn in the same manner and finally increment/multiplier `r` is drawn differently according to the chosen numerical scheme. More details on the drawing plan see Section 3.

Finally, function determines possible candidates for the clue of type `3` (see Subsection 2.1.17). Parameter `oper` in the clue of type `3` if either `'+'` or `'*'` so the function looks for both, additive and multiplicative clues regardless of the numerical scheme chosen earlier. For every possible clue candidate (for example `x=y+2`) function looks for the number of `x`'s and `y`'s satisfying the relation given in the clue candidate in chosen earlier set of numerical values. If there are at least `min_free` such pairs of `x` and `y` then clue candidate is appended to the final clue candidates list. `min_free` is defined by the equation:

$$\mathtt{min\_free} = \lfloor \frac{\mathtt{puzzle\_k} - 1}{2} \rfloor + 1$$

`min_free` is equal `2` for the `puzzle_k=3`, `3` for the `puzzle_k=5` and is generally larger for the larger values of `puzzle_k`.

Function returns tuple like the following:

```
('numerical', [1.0, 2.0, 4.0, 5.0],
  {'x=y*2', 'x=y*2.0', 'x=y+1.0', 'x=y+3.0'})
```

2.2.4 draw_num_interpretation

`draw_num_interpretation(values)` is an auxiliary external function. Function takes as an input numerical values of just one single numerical category drawn in the function `draw_num_scheme()` (see Subsection 2.2.3). Function basing on those values draw the interpretation for those values. Interpretations are hard-coded within this function. Every interpretation has its own conditions on the numerical values to satisfy and its own probability weight. Interpretations that do not satisfy those conditions for a given `values` are excluded from the draw. Then, draw takes place according to the probability weights of interpretations left. More details on those conditions and weights can be found in Section 3.

Returns tuple with drawn interpretation and the values like the following:

```
([1.0, 2.0, 4.0, 5.0], 'Nr_@')
```

2.2.5 draw_category

.

`draw_category(K, k, diff=3, seed=0)` is a collective external function. Function draws a full set of categories for the puzzle using functions described in the Subsections from 2.2.1 to 2.2.4.

Function set the random seed `seed`. Then, depending on the value of `diff` it chooses the probability distribution from which it will draw with replacement the category types (categorical, ordinal, numerical). More about those probabilities in the Section 3. Function guarantees that there always has to be at least one categorical and one numerical category. The rest of the categories are drawn according to the probability distribution by the functions describe in the previous subsections.

Functions guarantees that numerical interpretations won't repeat themselves (if it is possible). However it does not guarantee that categorical ones repeat (it is theoretically possible if there is the same object in more than one category). Therefore the method `draw_categories` of class `puzzle` protects against this event.

Function returns a list of drawn categories like the following:

```
[('categorical', ['Maja', 'Monika', 'Maria', 'Magdalena']),
 ('categorical', ['tarta', 'rolada', 'cannelloni', 'indyk']),
 ('ordinal', ['A4', 'A3', 'A2', 'A1']),
 ('numerical', [2, 10, 50, 250], {'x=y*25', 'x=y*5'}, '@_kg')]
```

2.2.6 get_string_name

`get_string_name(categories, K1, i1, replace_polish=False)` is an external function. Function returns the final object name that has to be displayed at the puzzle. Parameter `categories` is a whole set of categories in the form of the output of the `draw_category()` function. `K1` is the index of category and `i1` is the index of an object from that category. In the case of category being categorical or ordinal function simply returns the specified element from this category objects list. In the case of numerical category with the interpretation that contains the sign `@`, this sign is replaced by the specified object number and the resulting string is returned.

For example for the category:

```
 ('numerical', [2, 10, 50, 250], {'x=y*25', 'x=y*5'}, '@_kg')
```

and `i1=2` the resulting name would be "50 kg".

Additional feature to this function is governed by the parameter `replace_polish` which specifies whether Polish characters should be replaced. This useful in the case when the font used to printing do not support Polish characters.

2.2.7 do_categories_repeat

`do_categories_repeat(categories)` is an auxiliary external function. It is used in the method `draw_categories` of class `puzzle` to check if there are some repeating objects in the whole categories set (see Subsection 2.1.36). It compares the names created by the function `get_string_name()` (see Subsection 2.2.6). Therefore it is possible for two numerical objects to share the same number but have different interpretation. Returns boolean value.

2.3 pdf_printing_functions.py

Python packages required: `numpy`, `pandas`, `pathlib`, `os`, `reportlab`, `random`, `math`.

Uses functions from `generating_categories_functions.py` (see Section 2.2) and `puzzle_class.py` (see Section 2.1).

In this file all the functions for the needed to print logic puzzles into the pdf canvas are stored.

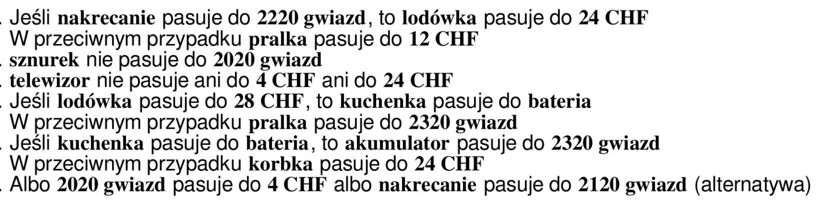
2.3.1 star

`star(canvas, xcenter, ycenter, radius)` is an auxiliary external function. Function draws on the canvas `canvas` at the coordinates `xcenter` and `ycenter` a star of size (radius) `radius`.

2.3.2 draw_clues_on_canvas

`draw_clues_on_canvas(categories, clue, c, X, Y, no, width)` is an auxiliary external function. Function draws on the canvas `c` one single clue `clue` at the coordinates `X` and `Y` and number `no`. Parameter `categories` is needed in order to encode the names of the categories objects involved in the clue. Parameter `width` specifies the font size. Parameter `no` describes the number that has to be printed at the beginning of the printed clue.

Object names are drawn in special font which is set to be `"Times-Bold"` whereas the rest of the clue is written in ”normal” font which is set to `"sans-serif"`. Font `"Times-Bold"` do not support all of the Polish characters and therefore replacement of Polish signs in needed (see Subsection 2.2.6). Examples of the clues are shown at the Figure 18.

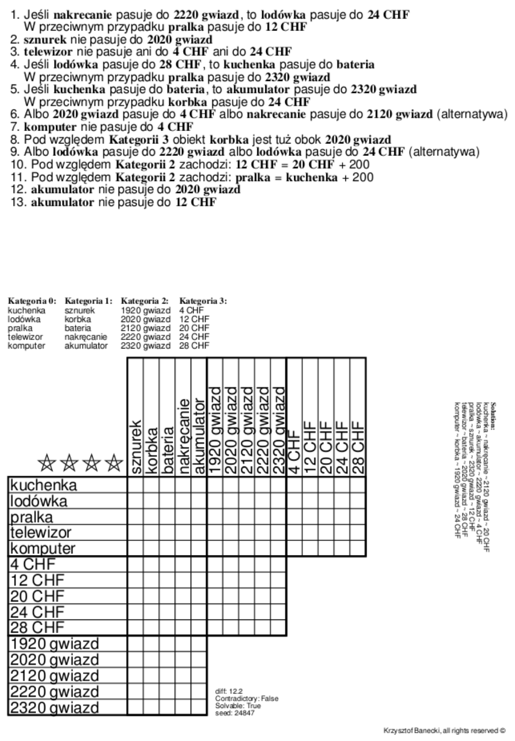


1. Jeśli **nakrecanie** pasuje do **2220 gwiazd**, to **lodówka** pasuje do **24 CHF**
   W przeciwnym przypadku **pralka** pasuje do **12 CHF**
2. **sznurek** nie pasuje do **2020 gwiazd**
3. **telewizor** nie pasuje ani do **4 CHF** ani do **24 CHF**
4. Jeśli **lodówka** pasuje do **28 CHF**, to **kuchenka** pasuje do **bateria**
   W przeciwnym przypadku **pralka** pasuje do **2320 gwiazd**
5. Jeśli **kuchenka** pasuje do **bateria**, to **akumulator** pasuje do **2320 gwiazd**
   W przeciwnym przypadku **korbka** pasuje do **24 CHF**
6. Albo **2020 gwiazd** pasuje do **4 CHF** albo **nakrecanie** pasuje do **2120 gwiazd** (alternatywa)

Figure 18: Example set of clues printed into pdf.

2.3.3 draw_on_canvas

`draw_on_canvas(puzzle1, c, X = 30, Y = 30, box_size = None)` is a collective external function. Function draws on the canvas `c` the whole information about the puzzle `puzzle1`. Example resulting canvas printed into `.pdf` file is presented in the Figure 19.



1. Jeśli **nakrecanie** pasuje do **2220 gwiazd**, to **lodówka** pasuje do **24 CHF**
   W przeciwnym przypadku **pralka** pasuje do **12 CHF**
2. **sznurek** nie pasuje do **2020 gwiazd**
3. **telewizor** nie pasuje ani do **4 CHF** ani do **24 CHF**
4. Jeśli **lodówka** pasuje do **28 CHF**, to **kuchenka** pasuje do **bateria**
   W przeciwnym przypadku **pralka** pasuje do **2320 gwiazd**
5. Jeśli **kuchenka** pasuje do **bateria**, to **akumulator** pasuje do **2320 gwiazd**
   W przeciwnym przypadku **korbka** pasuje do **24 CHF**
6. Albo **2020 gwiazd** pasuje do **4 CHF** albo **nakrecanie** pasuje do **2120 gwiazd** (alternatywa)
7. **komputer** nie pasuje do **4 CHF**
8. Pod względem **Kategorii 3** obiekt **korbka** jest tuż obok **2020 gwiazd**
9. Albo **lodówka** pasuje do **2220 gwiazd** albo **lodówka** pasuje do **24 CHF** (alternatywa)
10. Pod względem **Kategorii 2** zachodzi: **12 CHF = 20 CHF** + 200
11. Pod względem **Kategorii 2** zachodzi: **pralka = kuchenka** + 200
12. **akumulator** nie pasuje do **2020 gwiazd**
13. **akumulator** nie pasuje do **12 CHF**

Kategoria 0:	**Kategoria 1:**	**Kategoria 2:**	**Kategoria 3:**
kuchenka	sznurek	1920 gwiazd	4 CHF
lodówka	korbka	2020 gwiazd	12 CHF
pralka	bateria	2120 gwiazd	20 CHF
telewizor	nakręcanie	2220 gwiazd	24 CHF
komputer	akumulator	2320 gwiazd	28 CHF

diff: 12.2
Contradictory: False
Solvable: True
seed: 24847

Solution:
kuchenka - nakręcanie - 2120 gwiazd - 20 CHF
lodówka - akumulator - 2220 gwiazd - 4 CHF
pralka - sznurek - 2320 gwiazd - 12 CHF
telewizor - bateria - 2020 gwiazd - 28 CHF
komputer - korbka - 1920 gwiazd - 24 CHF

Krzysztof Banecki, all rights reserved ©

Figure 19: Example puzzle printed into pdf.

The height of the puzzle grid (squares for solving and rectangles with objects names) is `puzzle_h=400`. If the parameter `box_size` is not specified it is set to `puzzle_h/(K+0.5)`, where `K` equals `puzzle1.K`. This parameter specifies the length of the side of the single box. The length of the rectangles containing the objects names is always equal `1.5` times the box size.

Number of stars printed depends only of the `puzzle1.diff` variable. For `diff<2` 1 stars are printed, for `2<=diff<5` 2, for `5<=diff<10` 3 and 4 if `10<diff`.

Puzzle grid is printed starting at the bottom left point `(X,Y)` and assuming it has to have height of `puzzle_h`.

Clues are printed at the top of the page with function `draw_clues_on_canvas()` (see Subsection 2.3.2). Clues are randomly shuffled before printing. Parameter `N_lines` describes how many lines will those clues occupy. If this parameter is too big the font size is altered.

Diagnostic information is printed to the bottom right from the grid. It features the original `diff` value, values of the `contradictory` and `solvable` flags and random seed. In the case when the puzzle is either contradictory or not solvable the corresponding line will be printed in red font.

Solution to the puzzle is printed at the right side of the page. Categories descriptions are printed right above the puzzle grid. If Number of objects in each category `puzzle1.k` is greater than 5 not all of the objects are printed. Finally, objects names are typed into the puzzle grid.

2.4 statistics_functions.py

Python packages required: `numpy`, `pandas`, `pathlib`, `collections` and `matplotlib`.

In this file all the diagnostic functions for the needed useful to monitor the categories data base are stored. Those functions are additional and do not contribute to the generator operating process.

2.4.1 print_categories_stats

`print_categories_stats()` is an external diagnostic function. Function creates a single pie plot presenting the current distribution of probability weights across categories separately for every categorical class (activities, geography, items, names, nature, other concepts and special_names) and ordinal categories. Pie plots created by this function are presented in the Section 3.

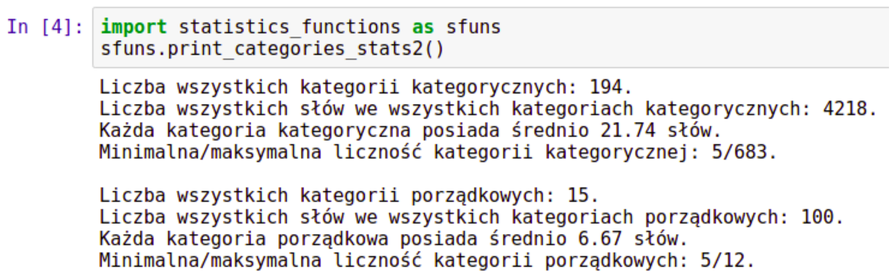
2.4.2 print_categories_stats2

`print_categories_stats2()` is an external diagnostic function. Function prints to the console basic diagnostic information about the whole categories data base. Both for categorical and ordinal categories it presents:

- Total number of categories.
- Total number of all the objects in all categories of certain type.
- Mean number of objects in each category.
- Minimal and maximal number of objects in a single category.

2.4.3 find_word

`find_word(word)` is an external diagnostic function. Function searches through all the categorical and ordinal categories in order to find the word `word`. Function prints the list of paths under which it found the given word or tell that that word (object name) does not appear in the current category data base.

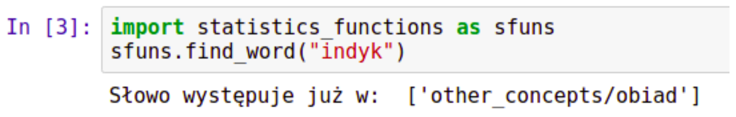


```
In [4]: import statistics_functions as sfuns
        sfuns.print_categories_stats2()

        Liczba wszystkich kategorii kategorycznych: 194.
        Liczba wszystkich słów we wszystkich kategoriach kategorycznych: 4218.
        Każda kategoria kategoryczna posiada średnio 21.74 słów.
        Minimalna/maksymalna liczność kategorii kategorycznej: 5/683.

        Liczba wszystkich kategorii porządkowych: 15.
        Liczba wszystkich słów we wszystkich kategoriach porządkowych: 100.
        Każda kategoria porządkowa posiada średnio 6.67 słów.
        Minimalna/maksymalna liczność kategorii porządkowych: 5/12.
```

Figure 20: Example output of the `print_categories_stats2` function.



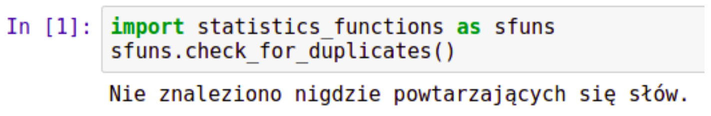
```
In [3]: import statistics_functions as sfuns
        sfuns.find_word("indyk")

        Słowo występuje już w:  ['other_concepts/obiad']
```

Figure 21: Example output of the `find_word` function.

2.4.4 check_for_duplicates

`check_for_duplicates()` is an external diagnostic function. Function searches through all the categorical and ordinal categories in order to category in which object names repeat. Function prints to the console the result of this search.



```
In [1]: import statistics_functions as sfuns
        sfuns.check_for_duplicates()

        Nie znaleziono nigdzie powtarzających się słów.
```

Figure 22: Example output of the `check_for_duplicates` function.

3 Drawing Schemes & Probabilities

In this chapter detailed description of the how the program draws everything for the puzzle generation in provided.

3.1 Categories drawing

Three kinds of categories are supported by the program: categorical, numerical and ordinal. Numerical categories are hard-coded in the `generating_categories_functions.py` (see Subsections 2.2.3 and 2.2.4). Categorical and ordinal categories are stored in the external folders in `.txt` files. All ordinal categories are stored in the `./categories/ordinal/` folder. All categorical categories are stored in the

`./categories/categorical/` folder. Categorical categories are then divided into 7 separate classes/folders, each of those classes containing some number of categories, each of category containing some number of objects. Each class of categories has its own probability weight assigned as it is shown at the below table:

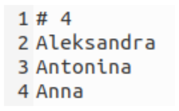
Category class	# of categories	Weight	Probability
activities	11	3	∼ 7.9%
geography	29	3	∼ 7.9%
items	44	5	∼ 13.2%
names	21	12	∼ 31.5%
nature	34	4	∼ 10.5%
other_concepts	52	5	∼ 13.2%
special_names	15	6	∼ 15.8%

Above probability weights can be changed at the beginning of the function `draw_cat_scheme()` (see Subsection 2.2.1).

In summary there are currently 206 categorical categories with around 22.9 objects in each category on average. Total number of categorical objects is 4716 although some may repeat themselves.

I the following subsections we present in detail statistics on the categorical categories in each folder. The last subsections are dedicated to ordinal and numerical categories.

Both categorical and ordinal categories have specific probability weights assigned to themselves. Those probabilities directly influence probability of drawing a certain category and are given in the following subsections in tables. Those weights can be altered directly in the `.txt` file where the category objects are being stored. The first line of every such file starts with "#" blank space and positive integer number which is the probability weight attached to this category. Every further line contains an object name from this category. See example at Figure 23, where the probability weight is 4 and first three objects names are: *"Aleksandra"*, *"Antonina"* and *"Anna"*.



```
# 4
Aleksandra
Antonina
Anna
```

Figure 23: Example beginning of the `.txt` file of a certain categorical (or ordinal) category.

When the categories are being drawn in function `draw_category()` the probabilities of whether the function chooses categorical, ordinal or numerical category depends on the parameter `diff`. Currently it is recommended to use `diff=3` (or do not use any because value `3` is set as default). In this case the distribution looks like:

- categorical - 47.6% (weight: 2)

- ordinal - 4.8% (weight: 0.2)
- categorical - 47.6% (weight: 2)

Probability weights can be altered at the beginning of the function `draw_category()` (see Subsection 2.2.5).

3.1.1 Categories from 'activities'

Statistics on categorical categories contained in folder:
`./categories/categorical/activities/`.
Probability of drawing this folder is around 7.9%.

Category	# of objects	Weight	Probability
czynnosci_biurowe	13	3	5.1%
czynnosci_domowe	8	6	10.2%
czynnosci_lazienkowe	7	4	6.8%
dzialania_matematyczne	7	5	8.5%
hobby	49	10	16.9%
hobby2	19	7	11.9%
hobby_ekstremalne	11	3	5.1%
majsterkowanie	18	4	6.8%
mowienie	106	3	5.1%
ruchowe	15	4	6.8%
sporty	43	10	16.9%

3.1.2 Categories from 'geography'

Statistics on categorical categories contained in folder:
`./categories/categorical/geography/`.
Probability of drawing this folder is around 7.9%.

Category	# of objects	Weight	Probability
dzielnice_warszawy	18	2	2.0%
jeziora_polski	10	7	6.9%
jeziora_swiata	9	1	1.0%
kontynenty	7	6	5.9%
kraje_afryka	18	2	2.0%
kraje_europa	26	10	9.8%
kraje_swiata	18	8	7.8%
lotniska_swiata	7	1	1.0%
miasta_polski_duze	19	5	4.9%
miasta_polski_male	21	3	2.9%
miasta_polski_wsie	18	4	3.9%
miasta_swiata	17	4	3.9%
miejsca_na_wypoczynek	10	8	7.8%

oceany	5	3	2.9%
parki_narodowe	23	2	2.0%
plemiona_indian	21	1	1.0%
rodzaje_gleb	7	2	2.0%
rodzaje_mineralow	9	1	1.0%
rodzaje_skal	10	2	2.0%
rzeki_polski_duze	10	7	6.9%
rzeki_polski_male	14	3	2.9%
rzeki_swiata	14	4	3.9%
stolice_europy	15	5	4.9%
stolice_swiata	12	2	2.0%
strefy_zycia_w_morzu	6	1	1.0%
szczyty_himalaje	13	1	1.0%
wojewodztwa	16	4	3.9%
wulkany_swiata	8	1	1.0%
wyspy_swiata	19	2	2.0%

3.1.3 Categories from 'items'

Statistics on categorical categories contained in folder:
`./categories/categorical/items/`.
Probability of drawing this folder is around 13.2%.

Category	# of objects	Weight	Probability
budynki_starozytne	12	1	1.0%
czesci_ogrodu	13	1	1.0%
czesci_samochodowe	39	1	1.0%
czesci_samochodowe2	14	2	1.9%
czesci_uzbrojenia	15	2	1.9%
domowe_agd	7	4	3.8%
elementy_komputera	11	3	2.9%
elementy_komputera2	8	1	1.0%
instrumenty_muzyczne	19	5	4.8%
kopalne	15	2	1.9%
meble	10	5	4.8%
narzedzia	14	5	4.8%
obiekty_kosmiczne	13	1	1.0%
obiekty_ukladu_slonecznego	18	1	1.0%
ozdoby	5	3	2.9%
pierwiastki_ciezkie	39	1	1.0%
pierwiastki_lekkie	19	1	1.0%
pierwiastki_szlachetne	6	1	1.0%

plac_zabaw	9	4	3.8%
planety	9	2	1.9%
pojazdy	21	5	4.8%
pojazdy_morskie	14	2	1.9%
pokoje_w_domu	11	4	3.8%
przebrania	13	2	1.9%
przedmioty_lazienka	16	3	2.9%
przedmioty_male	29	4	3.8%
przedmioty_naukowe	18	1	1.0%
przedmioty_ogrodowe	9	1	1.0%
przedmioty_silownia	10	1	1.0%
przyprawy	7	3	2.9%
rodzaje_maki	12	1	1.0%
rodzaje_podlog	6	1	1.0%
rodzaje_podloza	7	1	1.0%
rodzaje_rowerow	12	2	1.9%
rodzaje_soku	15	3	2.9%
rodzaje_zolnierzy	10	1	1.0%
rodzaje_zup	13	3	2.9%
ubrania	17	4	3.8%
ubrania_zimowe	11	2	1.9%
zabawki	12	5	4.8%
zabawy	13	2	1.9%
zmysly	5	5	4.8%
zwiazki_chemiczne	11	2	1.9%
zwiazki_chemiczne2	10	1	1.0%

3.1.4 Categories from 'names'

Statistics on categorical categories contained in folder: `./categories/categorical/names/`.

Probability of drawing this folder is around 31.5%.

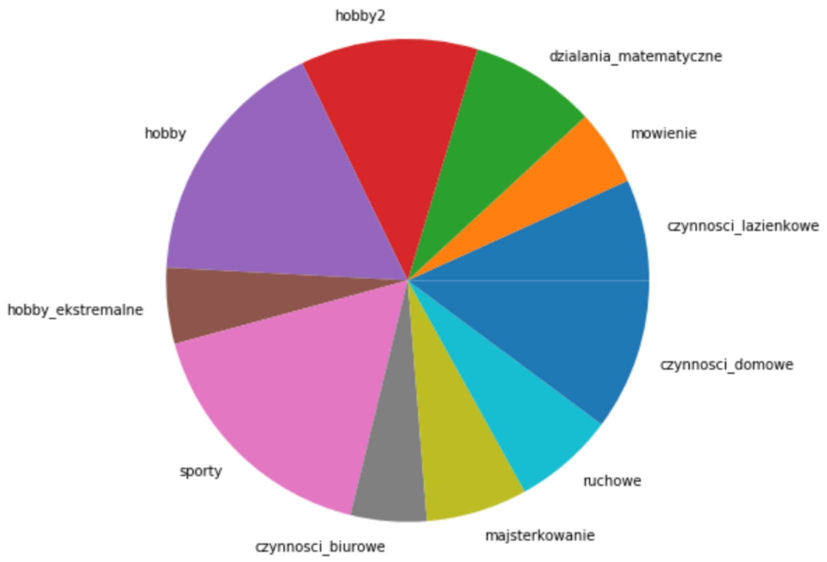


Figure 24: Pie chart of categories probabilities from 'activities' folder.

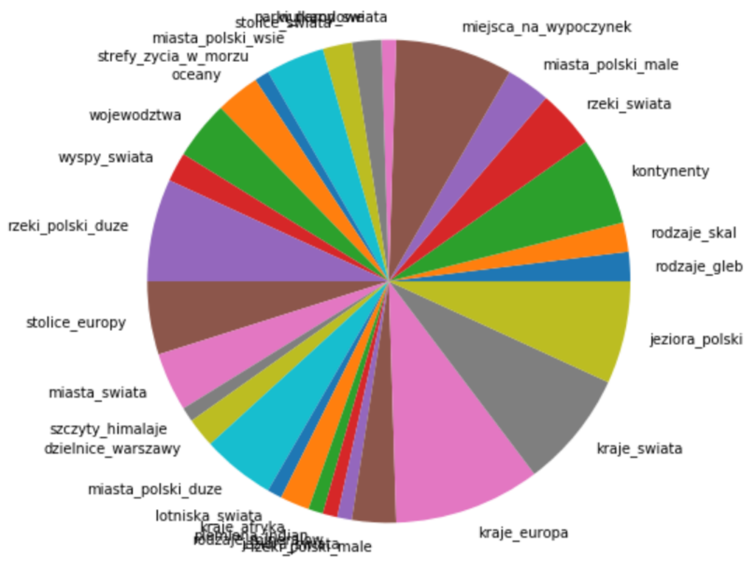


Figure 25: Pie chart of categories probabilities from 'geography' folder.

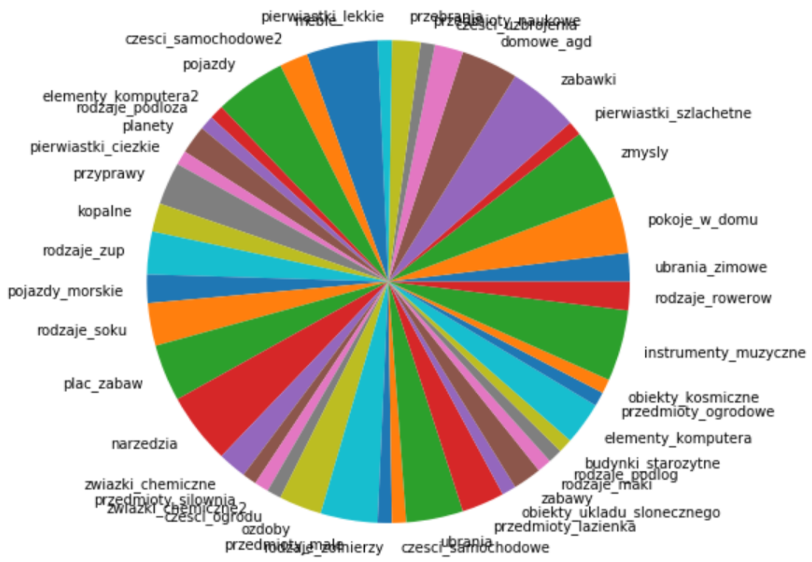


Figure 26: Pie chart of categories probabilities from 'items' folder.

Category	# of objects	Weight	Probability
imiona	49	10	18.2%
imiona_jan	8	2	3.6%
imiona_krolow	14	5	9.1%
imiona_rosyjskie_meskie	8	1	1.8%
imiona_rosyjskie_zenskie	15	1	1.8%
imiona_staropolskie_meskie	72	1	1.8%
imiona_staropolskie_zenskie	16	1	1.8%
imiona_zdrobniale	13	8	14.5%
imiona_zenskie_na_A	11	4	7.3%
imiona_zenskie_na_M	13	4	7.3%
nazwiska_afrykanskie	11	1	1.8%
nazwiska_angielskie	9	2	3.6%
nazwiska_arabskie	21	1	1.8%
nazwiska_chinskie	13	1	1.8%
nazwiska_francuskie	13	1	1.8%
nazwiska_hiszpanskie	12	1	1.8%
nazwiska_japonskie	14	1	1.8%
nazwiska_niemieckie	11	1	1.8%
nazwiska_polskie	20	7	12.7%
nazwiska_rosyjskie	14	1	1.8%
nazwiska_wloskie	12	1	1.8%

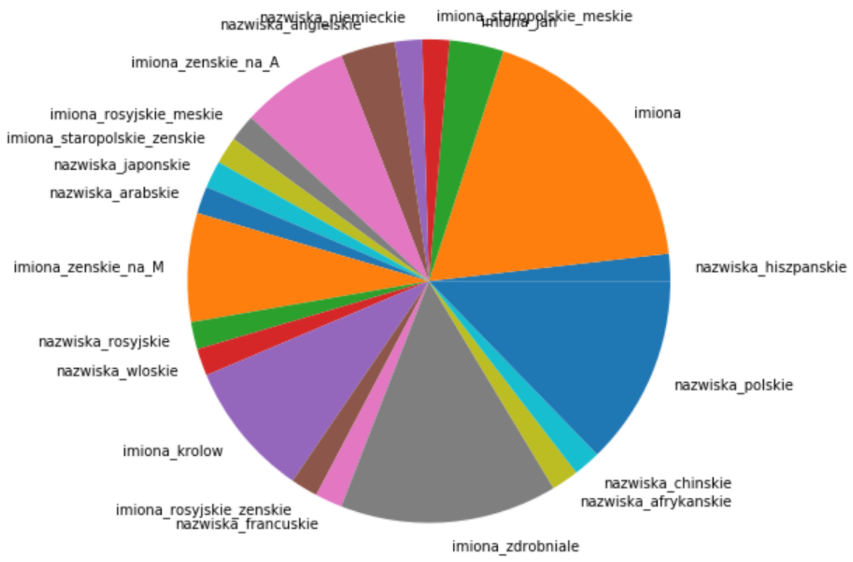


Figure 27: Pie chart of categories probabilities from 'names' folder.

3.1.5 Categories from 'nature'

Statistics on categorical categories contained in folder:
`./categories/categorical/nature/`.
Probability of drawing this folder is around 10.5%.

Category	# of objects	Weight	Probability
czesci_ciala	6	3	3.2%
czesci_ciala2	18	4	4.3%
czesci_ciala_wewnetrzne	10	2	2.2%
czesci_twarzy	12	4	4.3%
dinozaury	25	2	2.2%
drzewa_ogrodowe	15	4	4.3%
drzewa_tropikalne	8	1	1.1%
gatunki_czlowieka	6	1	1.1%
grzyby	11	4	4.3%
kosci	18	3	3.2%
kotowate	18	1	1.1%
kwiaty	25	6	6.5%
owoce	19	6	6.5%
owoce_tropikalne	13	2	2.2%
plazy	7	1	1.1%
ptaki_drapiezne	8	1	1.1%
ptaki_duze	10	2	2.2%

ptaki_krukowate	7	2	2.2%
ptaki_male	13	2	2.2%
rasy_psow	29	4	4.3%
typy_bezkregowcow	84	1	1.1%
typy_zwierzat	9	2	2.2%
uklad_krwionosny	6	1	1.1%
warzywa	41	6	6.5%
zboza	10	2	2.2%
ziola	12	3	3.2%
zwierzeta_arktyczne	8	1	1.1%
zwierzeta_domowe	9	7	7.5%
zwierzeta_dzikie	11	2	2.2%
zwierzeta_gospodarskie	7	7	7.5%
zwierzeta_malpy	7	2	2.2%
zwierzeta_morskie	10	2	2.2%
zwierzeta_owady	13	1	1.1%
zwierzeta_ryby	17	1	1.1%

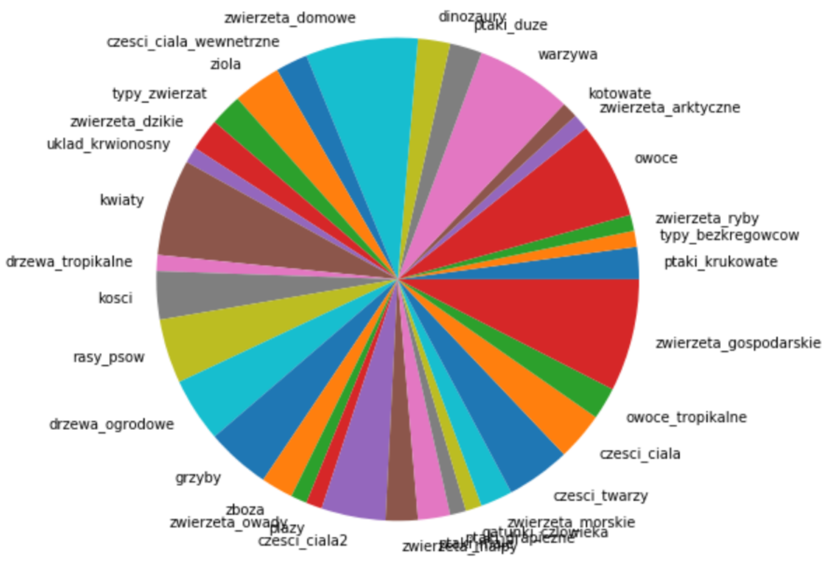


Figure 28: Pie chart of categories probabilities from 'nature' folder.

3.1.6 Categories from 'other_concepts'

Statistics on categorical categories contained in folder:

`./categories/categorical/other_concepts/`.
Probability of drawing this folder is around 13.2%.

Category	# of objects	Weight	Probability
cnoty	20	3	2.1%
czesci_mowy	10	1	0.7%
czesci_zdania	5	1	0.7%
deser	25	2	1.4%
dzialy_w_firmie	27	2	1.4%
figury_geometryczne	22	5	3.5%
formy_platnosci	7	4	2.8%
gatunki_filmowe	46	5	3.5%
gatunki_ksiazek	17	5	3.5%
harcerze	22	1	0.7%
kierunki_swiata	8	5	3.5%
kierunki_w_literaturze	30	3	2.1%
kierunki_w_malarstwie	14	3	2.1%
kolory_brazowe	18	1	0.7%
kolory_czerwone	15	1	0.7%
kolory_niebieskie	16	1	0.7%
kolory_rozowe	24	1	0.7%
kolory_teczy	8	9	6.3%
kolory_zielone	14	1	0.7%
kolory_zolte	13	1	0.7%
ministerstwa_w_polsce	20	1	0.7%
mityczne_stworzenia	42	2	1.4%
napoje	14	3	2.1%
obiad	47	4	2.8%
okazje	17	3	2.1%
okresy_w_historii_ziemi	15	1	0.7%
programy_telewizyjne	16	2	1.4%
przedmioty_studia	20	1	0.7%
przedmioty_szklane	17	1	0.7%
przedmioty_szkolne	15	6	4.2%
przymiotniki_ksztaltu	35	3	2.1%
przymiotniki_metalowe	16	4	2.8%
przymiotniki_polozenia	5	1	0.7%
przymiotniki_powazne	7	2	1.4%
przymiotniki_wielkosci	12	2	1.4%
relacje_rodzinne	28	7	4.9%
rodzaje_chmur	10	1	0.7%
rodzaje_komunikacji_miejskiej	9	4	2.8%
rodzaje_makaronu	20	1	0.7%
rodzaje_pierogow	12	1	0.7%
rodzaje_sera	20	1	0.7%

rodzaje_zasilania	7	3	2.1%
slowa_dziwne	38	1	0.7%
sniadanie	11	2	1.4%
sposoby_komunikacji	10	2	1.4%
sposoby_przyrzadzania_jajka	12	1	0.7%
tematy_przyjecia	21	2	1.4%
uczucia_negatywne	85	3	2.1%
uczucia_pozytywne	69	3	2.1%
zawody	67	10	7.0%
zrodla_energii	9	3	2.1%
zywioly	9	6	4.2%

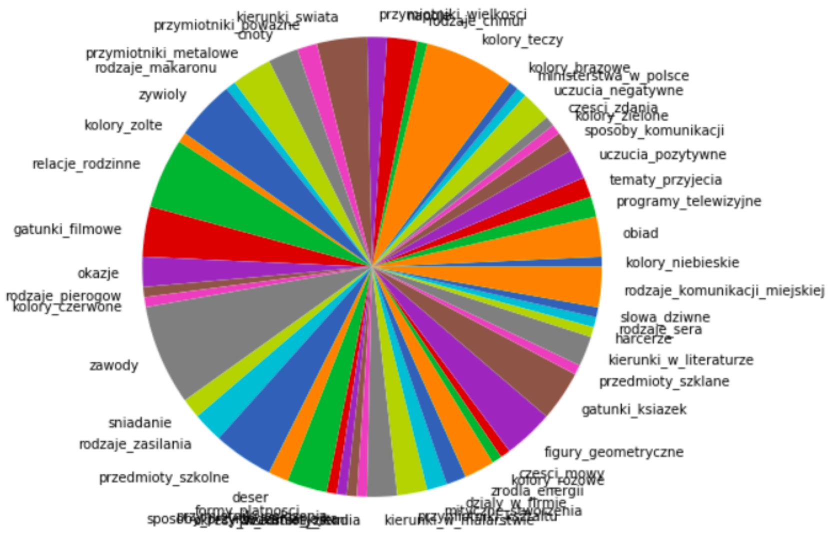


Figure 29: Pie chart of categories probabilities from 'other_concepts' folder.

3.1.7 Categories from 'special_names'

Statistics on categorical categories contained in folder:
`./categories/categorical/special_names/`.
Probability of drawing this folder is around 15.8%.

Category	# of objects	Weight	Probability
dziwne_imiona_meskie	17	1	2.3%
dziwne_imiona_zenskie	18	1	2.3%
firmy_elektronika_z_X	14	1	2.3%
firmy_owocowe_z_pol	17	1	2.3%
imiona_dla_kota	683	10	23.3%
imiona_dla_psa	398	10	23.3%
imiona_finskie	88	2	4.7%
imiona_z_olo	10	1	2.3%
logi_zagadki	7	4	9.3%
nazwiska_od_drzew	18	1	2.3%
nazwiska_od_gryzoni	11	1	2.3%
nazwiska_od_panstw	79	4	9.3%
nazwiska_od_wojewodztw	13	3	7.0%
zmysleni_superbohaterowie_ang	34	1	2.3%
zmysleni_superbohaterowie_pol	29	2	4.7%

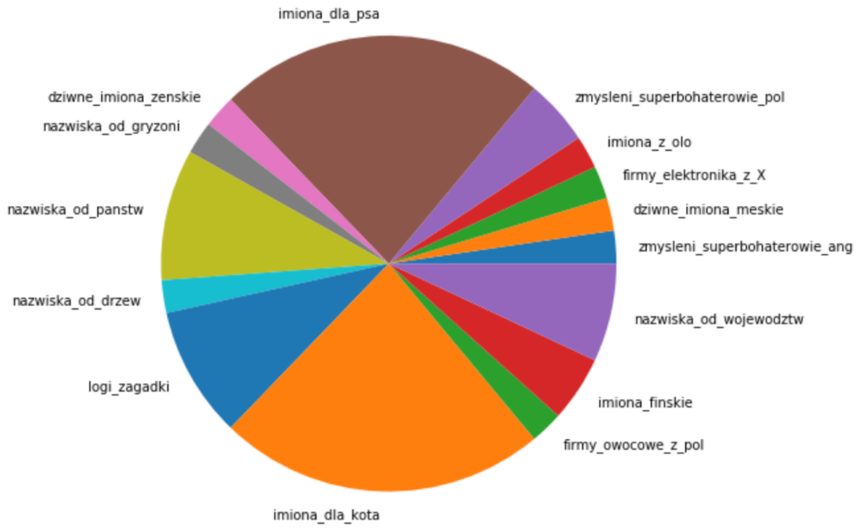


Figure 30: Pie chart of categories probabilities from 'special_names' folder.

3.1.8 Ordinal categories

In this subsection statistics on ordinal categories contained in folder `./categories/ordinal/` are presented. In summary there are currently 15 ordinal categories with around 6.7 objects in each category on average. Total number of ordinal objects is 100.

Category	# of objects	Weight	Probability
bajty	6	1	1.2%
dni_tygodnia	7	10	12.0%
formaty_kartki	7	4	4.8%
grubosc	5	5	6.0%
jednostki_dlugosci	6	3	3.6%
jednostki_masy	6	2	2.4%
krainy_geogr	5	4	4.8%
miesiace	12	10	12.0%
ocena_jakosci	7	8	9.6%
ocena_slownie	5	8	9.6%
rozmiar_ubrania	8	7	8.4%
temperatura	7	5	6.0%
uczucie	6	4	4.8%
wielkosc	8	7	8.4%
wysokosc	5	5	6.0%

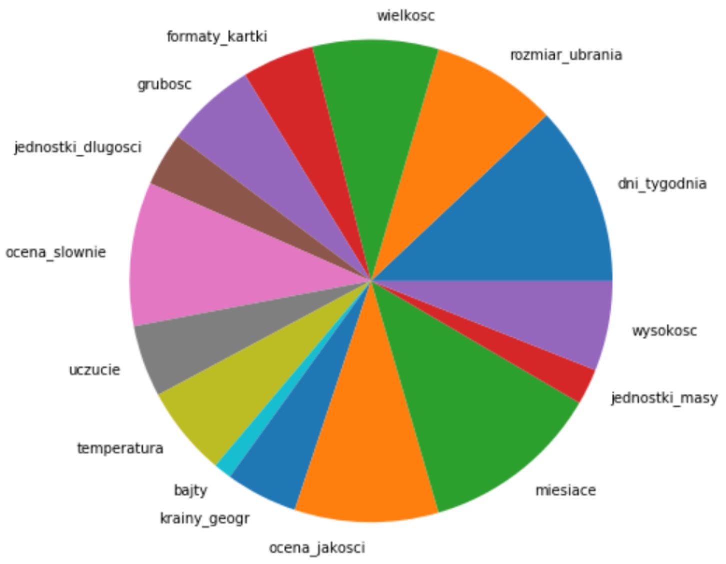


Figure 31: Pie chart of ordinal categories probabilities.

3.1.9 Numerical categories - values

Sampling numerical categories is divided into two parts. Firstly, numerical values are drawn and secondly interpretation of those numbers is being drawn to those values. In this subsection we present how numerical values are drawn. Interpretations are described in the next subsection.

Numbers of the numerical category are being drawn in the `draw_num_scheme()` function. At the beginning, general scheme of the numerical values is chosen according to the probability weights presented in the following table.

Numerical scheme	Weight	Probability	Example (for `k=6`)
arithmetic sequence	10	62.5%	$1, 2, 3, 4, 5, 6$
geometric sequence	2	12.5%	$1, 2, 4, 8, 16, 32$
ascending sequence	4	25%	$1, 2, 4, 5, 7, 8$

Those probability weights can be changed at the beginning of the function `draw_num_scheme()`.

Regardless of the numerical scheme, specific number are drawn basing on three factors: base value `p`, times factor `n` and increment/multiplier `r`.

- If the arithmetic sequence is chosen final numerical values will be:

$$\mathtt{p} \cdot \mathtt{n}, (\mathtt{p} \cdot \mathtt{n}) + \mathtt{r}, (\mathtt{p} \cdot \mathtt{n}) + 2\mathtt{r}, (\mathtt{p} \cdot \mathtt{n}) + 3\mathtt{r}, \ldots$$

- If the geometric sequence is chosen final numerical values will be:

$$\mathtt{p} \cdot \mathtt{n}, (\mathtt{p} \cdot \mathtt{n}) \cdot \mathtt{r}, (\mathtt{p} \cdot \mathtt{n}) \cdot \mathtt{r}^2, (\mathtt{p} \cdot \mathtt{n}) \cdot \mathtt{r}^3, \ldots$$

- If the ascending sequence is chosen final numerical values will look similar to the arithmetic sequence but from time to time `2r` will be added instead of `r`. Additionally, increment `r` is always equal to $\mathtt{n} \cdot \mathtt{p}$ in this case.

Regardless of the numerical scheme factors `p` and `n` are drawn from the probability described in the tables below.

p	Weight	Probability
−40	1	1.0%
−10	1	1.0%
−5	1	1.0%
0	1	1.0%
1	20	19.4%
1.5	5	4.9%
2	10	9.7%
2.5	5	4.9%
3	4	3.9%
4	4	3.9%
5	3	2.9%
6	3	2.9%
7	2	1.9%
8	2	1.9%
9	2	1.9%
10	5	4.9%

20	5	4.9%
50	5	4.9%
100	6	5.8%
1000	6	5.8%
1500	4	3.9%
1920	1	1.0%
1950	1	1.0%
1980	1	1.0%
1990	1	1.0%
2000	4	3.9%

`n`	Weight	Probability
1	10	66.7%
2	4	26.7%
3	1	6.6%

Factor `r` is drawn differently for every numerical scheme. For the "ascending sequence" it is not drawn at all. In this case difference between numerical value is equal either to the first value (`n*p`) or double that value (`2*n*p`) with probabilities 80% and 20% respectively.

Probabilities of a certain factor `r` values for "arithmetic sequence" scheme and "geometric sequence" scheme are given in the tables below. In "arithmetic sequence" factor `r` is interpreted as increment, whereas in the "geometric sequence" as a multiplier.

`r` (as increment)	Weight	Probability
0.5	10	6.4%
1	40	25.6%
1.5	5	3.2%
2	15	9.6%
2.5	5	3.2%
3	5	3.2%
4	5	3.2%
5	10	6.4%
10	10	6.4%
15	4	2.6%
25	10	6.4%
50	10	6.4%
100	8	5.1%
200	6	3.8%
250	3	1.9%
500	5	3.2%
1000	5	3.2%

`r` (as multiplier)	Weight	Probability
0.5	10	29.4%
2	10	29.4%
3	5	14.7%
4	4	11.8%
5	3	8.8%
10	2	5.9%

Additional preventive limitations are also taken there to avoid absurd values. When factor `r` in an increment ("arithmetic sequence" scheme):

- If the first element of a sequence is small (that is `<10` and `>-5`) only small (`<=10`) increments are possible.
- If the first element of a sequence is huge but cannot have an interpretation of year (`>100` and (`<=1500` or `>=2300`) increment also has to be big (`>=25`).
- If the first element of a sequence is big (`>=20`) then increment has to be integer value.

When factor `r` in an multiplier ("geometric sequence" scheme):

- If the first element of a sequence is not divisible by $2^{(k-1)}$ multiplier cannot be smaller than 1. This restriction aims to prevent from too small fractions as numerical values.
- If the first element of a sequence is not divisible by 100 and bigger than 50 then multiplier can only be either 2 or 10.

All of the probability weights described in this subsection can be altered in function `draw_num_scheme()` (see Subsection 2.2.3).

3.1.10 Numerical categories - interpretations

Drawing numerical interpretations is done in function `draw_num_interpretation()` (see Subsection 2.2.4). Drawing depends solely on the numerical values drawn by `draw_num_scheme()` (see Subsection 2.2.3). Every interpretation has its own conditions on the numerical values. If values do not satisfy condition of a certain interpretation this interpretation cannot be attached to those numerical values. The table below presents the complete list of numerical interpretations, their probability weights along with their conditions on numerical values. Abbreviation legend is provided right after the table.

Numerical interpretation	Weight	Conditions
"@ rok/lata"	35	int, +
"@ klocków"	5	int, +
"@ punktów"	10	int, +
"@ okruszków"	5	int, +

”@ ziaren”	5	int, +
”@ kropelek”	5	int, +
”@ groszków”	5	int, +
”@ plamek”	5	int, +
”@ drobinek”	5	int, +
”@ liści”	5	int, +
”@ kapsli”	5	int, +
”@ pocztówek”	3	int, +
”@ kulek”	5	int, +
”@ muszelek”	5	int, +
”@ drzew”	5	int, +
”@ kwiatów”	5	int, +
”@ owadów”	5	int, +
”@ ton”	3	int, +
”@ schodów”	5	int, +
”@ domów”	5	int, +
”@ ludzi”	5	int, +
”@ gwiazd”	4	int, +
”@ puzzli”	2	int, +
”@ zł”	35	int, +
”@ EUR”	15	int, +
”@ USD”	15	int, +
”@ CHF”	5	int, +
”@ denarów”	2	int, +
”@ koron”	5	int, +
”@ rok”	100	$\text{int}, < 2040, > 1300$
”@ stopni F”	5	$< 120, > -460$
”@ stopni C”	5	$< 100, > -273$
”@ pasków”	5	$\text{int}, +, < 50$
”@ misiów”	5	$\text{int}, +, < 50$
”@ lalek”	5	$\text{int}, +, < 50$
”@ miejsce”	15	$\text{int}, +, < 50$
”Nr @”	15	$\text{int}, +, < 50$
”@ dni”	15	$\text{int}, +, < 50$
”@ miesięcy”	5	$\text{int}, +, < 50$
”kanał @”	5	$\text{int}, +, < 50$
”@ kg”	15	+
”@ W”	5	+
”@ J”	5	+
”@ l”	15	$+, < 1000$
”@ m”	15	$+, < 1000$
”@ s”	15	$+, < 1000$
”@ szklanki”	15	$+, < 11$
”@ łyżeczki”	10	$+, < 11$
”@ wiaderka”	5	$+, < 11$
”@ koszy”	5	$+, < 11$

”@ filiżanek”	5	$+, < 11$
”@ %”	10	$> -0.01, < 100.01$
”@ stron”	10	int, $+, < 600$
”@ MHz”	5	int, $+, < 600$
”@ warianty”	5	int, $+, < 10$
”@-osobowy”	15	int, $+, < 10$
”@ pasażerów”	10	int, $+, < 10$
”Rozpoczęcie”	5	int, $+, < 24$
”Zakończenie”	5	int, $+, < 24$
”Sprawdzian”	5	int, $+, < 24$
”@ interwencji”	5	int, $+, < 24$
”@:00”	25	int, $+, < 24$
”@ sztuk”	5	int, $+, < 24$
”@ groszy”	5	int, $+, < 24$
”zysk/strata”	25	< 50
”Kolejność”	55	int, $+, < 11, 1 \in$
”Wynik”	5	
”Liczba”	5	
”Pomiar”	5	
”Wartość stałej”	5	

Legend for condition abbreviations:

- ”int” - all values have to be integers.
- ”$+$” - all values have to be positive.
- ”$-$” - all values have to be negative.
- ”$> x$” - all values have to be greater than x.
- ”$< x$” - all values have to be smaller than x.
- ”$x \in$” - values have to contain x.

All numerical interpretations, their probability weights and conditions can be altered in `draw_num_interpretation()` function (see Subsection 2.2.4).

3.2 Clues drawing

Six types of clues are currently supported. Those clues are in more detail described in Subsections from 2.1.15 to 2.1.20. Clues are being drawn by the method `draw _clues()` (see Subsection 2.1.37).

Firstly, clues of all the types other than `1` are sampled. Number of such clues is determine by the equation:

$$\text{Number of non-1 clues } = \lceil k \cdot K/2.3 \rceil. \qquad (2)$$

Non-1 clues are drawn with equal probability for every clue type: 20%. Clues of type 1 are then drawn until the program is able to solve the puzzle or the puzzle is contradictory.

4 Graphical User Interface

Graphical side of the program is held by the script `app_v1_1.py`. In Unix systems with `Python 3` and `tkinter` package installed this script can be called directly from the terminal. The Windows compiled version of this script can be accessed from the `app_v1_1.exe` file compiled via `pyinstaller` package. Details about the compilation of a program for Windows machines is described in Section 4.2. Detailed informations about the GUI itself are provided in Section 4.1.

4.1 app_v1_1.py

'Introduction frame' to the application consists of the three labels:

- "Welcome to my text puzzle generator!"
- "This is version 1.1."
- "Choose what you want to do:"

'Choosing frame' consist of radio button with two labels: "Generate random puzzle" and "Generate from seed". The function `chosen_random()` is called when first option is clicked, and function `chosen_seed()` when second is clicked. One text field is presented to enter the given random seed.

'Specifying sizes frame' consists of the label "Specify dimensions for your puzzle:" and two other labels: "Number of categories(> 2): " and "Number of objects in each category(> 2):" with their text entries for the dimensions of the puzzle.

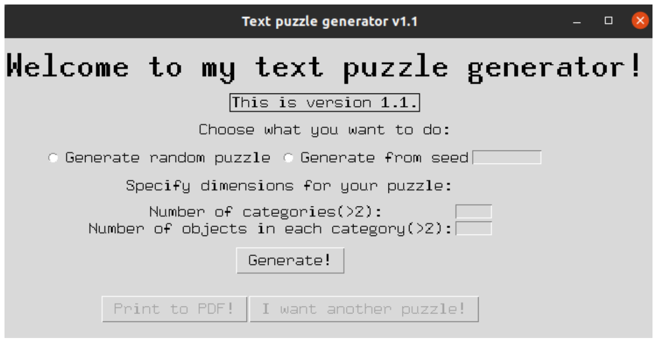
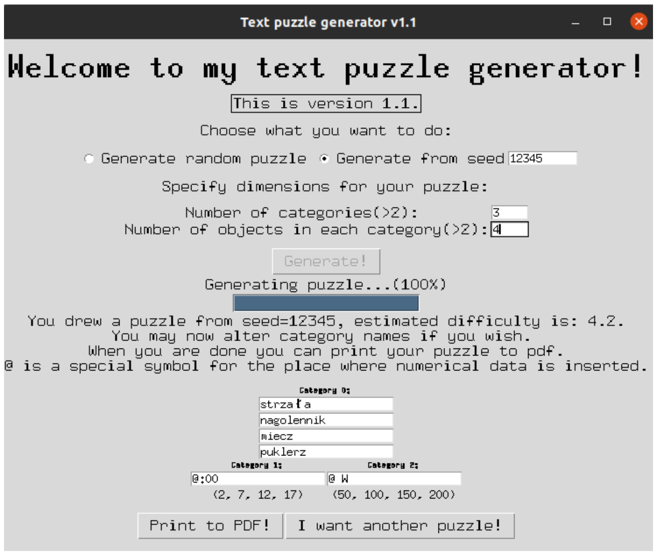

Text puzzle generator v1.1
Welcome to my text puzzle generator!
This is version 1.1.
Choose what you want to do:
Generate random puzzle
Generate from seed
Specify dimensions for your puzzle:
Number of categories(>2):
Number of objects in each category(>2):
Generate!
Print to PDF!
I want another puzzle!

Figure 32: Application start view.

'Generate frame' consists of the button "Generate!" label "Generating puzzle...(0%)" and progress bar. Last two elements are exposed only after the Generate button has been clicked. Function `clicked_gen()` is called after the button is clicked. The button is disabled then until the "I want another puzzle!" button is clicked.

Function `clicked_gen()` is firstly checking if the random seed and puzzle dimensions are provided in a proper form and returns warning otherwise. If all conditions are satisfied for the entry data the generation of the puzzle begins. Function is recreating the exact steps of puzzle generation described in Subsection 2.1.41. The only difference is that after each major step progress bar is being updated. Function then fills the 'Info about generated puzzle frame' and enables 'Final buttons frame' buttons.



Welcome to my text puzzle generator!

This is version 1.1.

Choose what you want to do:

Generate random puzzle [✓ Generate from seed] 12345

Specify dimensions for your puzzle:

Number of categories(>2): 3
Number of objects in each category(>2): 4

Generating puzzle...(100%)

You drew a puzzle from seed=12345, estimated difficulty is: 4.2.
You may now alter category names if you wish.
When you are done you can print your puzzle to pdf.
@ is a special symbol for the place where numerical data is inserted.

Category 0:
strzała
nagolennik
miecz
puklerz

Category 1: @:00
(2, 7, 12, 17)

Category 2: @ W
(50, 100, 150, 200)

Figure 33: Application view after generating the puzzle.

'Info about generated puzzle frame' consists of label `lbl7` including the information about the just sampled puzzle and two frames: first for categorical and ordinal puzzle objects and second for numerical categories. Category objects are displayed in entry fields so they can be altered before printing. Categorical objects can be changed in any way except object cannot repeat themselves. Ordinal objects also can be changed freely but one should keep in mind not to violate the relationship between ordinal objects. Otherwise the puzzle may be broken. In the case of numerical categories only numerical interpretation can be changed since numbers may play an important role in the way the puzzle is solved. The sign `@` means the numerical data will be inserted in its place in the pdf file. If numerical interpretation doesn't contain the `@` sign, it will be printed once on the top of a category box (not repeatedly in the name of each object). If the numerical interpretation is empty (which is also allowed as long no object repeat itself in the puzzle) none of the above is printed.

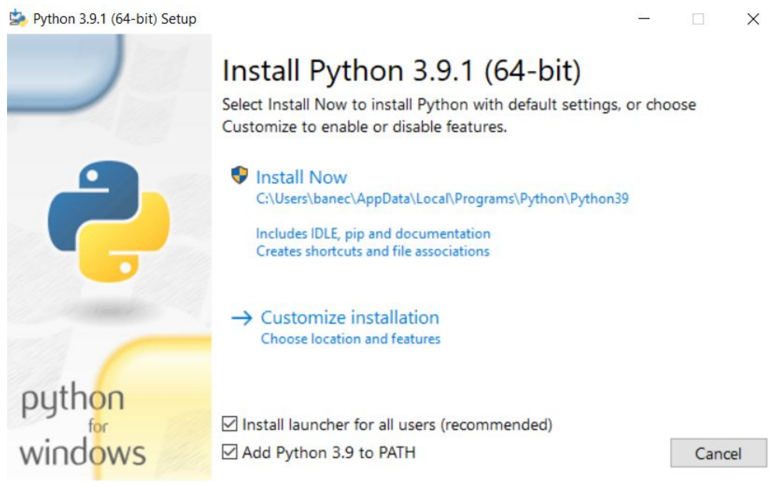
'Final buttons frame' consists of two buttons "Print to PDF!" and "I want another puzzle!". First one calls the `clicked_print()` function and starts the printing procedure. The user is asked to choose the destination folder for the pdf file and file is created by the program. The second button calls the function `clicked_new()` function, the program forgets about the current puzzle and enables the Generate button again.

'Final info frame' consists of one single label `final_lbl` which hold the info of the just generated and printed puzzle's name and destination folder.

4.2 Instructions on compiling for Windows

On Unix machines the above GUI may be run directly from the `app_v1_1.py` file. In order to compile this script to an `.exe` file readable for Windows systems via `pyinstaller` package, below steps may have been taken.

Firstly, install `Python` on your computer. Go to the `python.org` website and download the version `3+`. You'll need to add `Python` to PATH variable. The easiest way to do that is to mark "Add Python 3.x to PATH" during installation. Enable to change PATH length limit if needed during installation.


Python 3.9.1 (64-bit) Setup

Install Python 3.9.1 (64-bit)

Select Install Now to install Python with default settings, or choose Customize to enable or disable features.

Install Now
C:\Users\banec\AppData\Local\Programs\Python\Python39

Includes IDLE, pip and documentation
Creates shortcuts and file associations

Customize installation
Choose location and features

☑ Install launcher for all users (recommended)
☑ Add Python 3.9 to PATH

Figure 34: Add Python to path during installation.

In the 'App execution aliases' turn off the linkage between `python.exe` and `python3.exe` and App Installer.

Open the command line and type `python --version` to see if it is already added to the PATH (if you have opened it before close and reopen). If you get message like `"'python' is not recognized as an internal or external command"` then it is probably not added to the PATH. In this case it was probably installed in the

`C:\Users\(Your logged in User)\AppData\Local\Programs\Python\Python39` folder. Change "39" if you have downloaded a different version.

To add this path to the PATH variable, find 'Edit the system environmental variables' in the Control panel. Navigate to Advanced-¿Environment Variables and add the above path as well as the \Python39\Scripts path to your path variable separated by ";".

If the command `python --version` prints something like `Python 3.9.1` now, Python is successfully installed.

You need to install some crucial packages now. Before installing you may need to upgrade your `pip` by a command: `python -m pip install --upgrade pip`. We will need to install the following packages: `numpy`, `pandas`, `reportlab` and `pyinstaller`. Type the following commands:

- `python -m pip install numpy`
- `python -m pip install pandas`
- `python -m pip install reportlab`
- `python -m pip install pyinstaller==4.1` (You may face some issues with detecting your file as a virus with newer versions of `pyinstaller`)

Specify paths where you store the `.py` scripts and `categories` folder in `app.spec` file as follows:

```
 added_files = [
    ('C:\\path\\to\\your\\folder\\categories','data')
]

a = Analysis(['app_v1_1.py'],
             pathex=['C:\\path\\to\\your\\folder'],
             ...
```

Remember about double "\" signs in the path specifications.

Open the command line again. Navigate to the folder with all of the following scripts and files:

- `app_v1_1.py`
- `puzzle_class.py`
- `generating_categories_functions.py`
- `pdf_printing_functions.py`
- `app.spec`
- `categories` folder

Type into the command line `pyinstaller app.spec`. The resulting program: `app.exe` will be created in the folder `dist`. The last thing you need to do is to copy the `categories` folder into the `dist` folder. The application can now be started by double clicking on the `app.exe` file.

One last note: before running `pyinstaller app.spec` you may also need to change the default font used by the program to write on the `.pdf` file. To do that replace the line:

```
pdfmetrics.registerFont(TTFont('sans-serif', 'FreeSans.ttf'))
```

in `pdf_printing_functions.py` with:

```
pdfmetrics.registerFont(TTFont('sans-serif', 'times.ttf'))
```

This font should work just fine on Windows.

5 Examples

In this chapter some simple examples of how to use the objects of class `puzzle`.

5.1 Example 1

Let us import the `puzzle_class` module and initialize the object `puzzle1` of class `puzzle`. We would go for the puzzle with number of categories `K=4` and number of objects in each category `k=5`.

```
In [1]: from puzzle_class import puzzle

In [2]: puzzle1 = puzzle(K=4, k=5)
```

We may now print the basic information about this object by method `print_info()`.

```
In [3]: puzzle1.print_info()

        Seed: 0, difficulty: 0
        Completed: False, Contradictory: False
        K: 4, k: 5, No of clues: 0[0, 0, 0, 0, 0, 0]

In [4]: puzzle1.clues

Out[4]: []
```

As we can see the seed is initialized with 0, difficulty is set to 0, flags indicating whether the puzzle is completed and contradictory are both set to `False`. Both the list of categories and the list of clues are empty.

Let us try to draw something. We set the random seed to 12345 and draw categories by method `draw_categories()`.

```
In [5]: puzzle1.set_seed(12345)
        puzzle1.draw_categories()

In [6]: puzzle1.categories

Out[6]: [('categorical', ['piła', 'kombinerki', 'gwóźdź', 'łom', 'scyzoryk']),
         ('categorical',
          ['Piotrowski', 'Kowalski', 'Nowak', 'Lewandowski', 'Iksiński']),
         ('numerical',
          [40.0, 120.0, 200.0, 280.0, 360.0],
          {'x=y+160.0', 'x=y+80.0'},
          '@ m'),
         ('numerical', [1.5, 4.0, 6.5, 9.0, 11.5], {'x=y+2.5', 'x=y+5.0'}, '@ s')]
```

As we can see we have drawn two categorical and two numerical categories. We may now want to draw some clues by method `draw_clues()` in order to make the valid puzzle to solve.

```
In [7]: puzzle1.draw_clues()
```

```
In [8]: print(puzzle1.clues)
```

```
[{'typ': 4, 'K1': 2, 'i1': 4, 'K2': 3, 'i2': 1, 'K3': 0, 'i3': 3, 'K4': 1, 'i4': 3, 'K5': 0, 'i5': 0, 'K6': 1, 'i
6': 4}, {'typ': 6, 'K1': 3, 'i1': 0, 'K2': 3, 'i2': 1, 'K6': 2}, {'typ': 5, 'K1': 1, 'i1': 2, 'K2': 2, 'i2': 0, 'K
3': 0, 'i3': 1, 'K4': 3, 'i4': 3}, {'typ': 4, 'K1': 1, 'i1': 2, 'K2': 3, 'i2': 1, 'K3': 2, 'i3': 1, 'K4': 3, 'i4':
2, 'K5': 2, 'i5': 1, 'K6': 3, 'i6': 0}, {'typ': 5, 'K1': 1, 'i1': 3, 'K2': 3, 'i2': 1, 'K3': 1, 'i3': 0, 'K4': 3,
'i4': 2}, {'typ': 3, 'K1': 2, 'i1': 0, 'K2': 1, 'i2': 4, 'K6': 3, 'diff': 5.0, 'oper': '+'}, {'typ': 6, 'K1': 1,
'i1': 1, 'K2': 1, 'i2': 0, 'K6': 3}, {'typ': 2, 'K1': 1, 'i1': 1, 'K2': 1, 'i2': 3, 'K3': 3, 'i3': 4, 'K6': 2},
{'typ': 1, 'K1': 1, 'i1': 2, 'K2': 2, 'i2': 1}]
```

So, we have drawn 9 clues. We can check what types of clues are those again by the `print_info()` method.

```
In [9]: puzzle1.print_info()
```

```
Seed: 12345, difficulty: 0
Completed: False, Contradictory: True
K: 4, k: 5, No of clues: 9[1, 1, 1, 2, 2, 2]
```

So we have one clue of each of the types 1, 2 and 3, and two clues of each of the types 4, 5 and 6. Unfortunately, our puzzle is contradictory now, so in order to crate a valid puzzle we would have to draw the categories again.

5.2 Example 2

In this example we will use the recommended `generate()` method to generate puzzle. The function will automatically draw clues again if the resulting puzzle was contradictory. We would use the same dimensions and random seed as it was presented in the previous example.

```
In [1]: from puzzle_class import puzzle
```

```
In [2]: puzzle1 = puzzle(K=4, k=5)
        puzzle1.generate(seed=12345, trace=True)
```

```
Generating puzzle for seed=12345
Drawing categories...
Categories drawn:
('categorical', ['piła', 'kombinerki', 'gwóźdź', 'łom', 'scyzoryk'])
('categorical', ['Piotrowski', 'Kowalski', 'Nowak', 'Lewandowski', 'Iksiński'])
('numerical', [40.0, 120.0, 200.0, 280.0, 360.0], {'x=y+80.0', 'x=y+160.0'}, '@ m')
('numerical', [1.5, 4.0, 6.5, 9.0, 11.5], {'x=y+2.5', 'x=y+5.0'}, '@ s')
Drawing clues...
No of clues drawn = 10[1, 2, 3, 0, 3, 1]
Restricting clues...
Restricting clue 1/10, type: 1
Restricting clue 2/10, type: 5 OUT
Restricting clue 3/10, type: 6
Restricting clue 4/10, type: 3
Restricting clue 5/10, type: 3
Restricting clue 6/10, type: 2
Restricting clue 7/10, type: 3
Restricting clue 8/10, type: 5 OUT
Restricting clue 9/10, type: 2
Restricting clue 10/10, type: 5
Final difficulty assessment...
Difficulty: 1.8[2, 1, 2, 2, 2]
```

We called methdo `generate()` with the parameter `trace=True`. Thanks to this additional, diagnostic information is printed. If one don't want to look at it, the function can be run with `trace=False`. As we can see, we drawn exactly the same categories as in previous example, but this time the program has drawn different clues to them in the number of 10. Then the program started to trying throw out some of the clues, starting fro the type 1 clues. Finally, program manage to solve the puzzle without two type 5 clues and set the final clues set of size 8. The last thing the method performs is difficulty assessment which turned out to be 1.8, which is pretty easy and will be assigned with only one star (although difficulty may differ slightly for every run, even for the same random seed).

Finally, we can print our generated puzzle into the `.pdf` file. We import the needed `reportlab` package, our module `pdf_printing_functions.py` and print the puzzle by function `draw_on_canvas()`.

In [3]:
```
from reportlab.pdfgen import canvas
from pdf_printing_functions import draw_on_canvas
```

In [4]:
```
c = canvas.Canvas("testy.pdf")
draw_on_canvas(puzzle1, c)
c.showPage()
c.save()
```

Final `.pdf` file containing our puzzle have been just created in the current folder under the name `testy.pdf`.