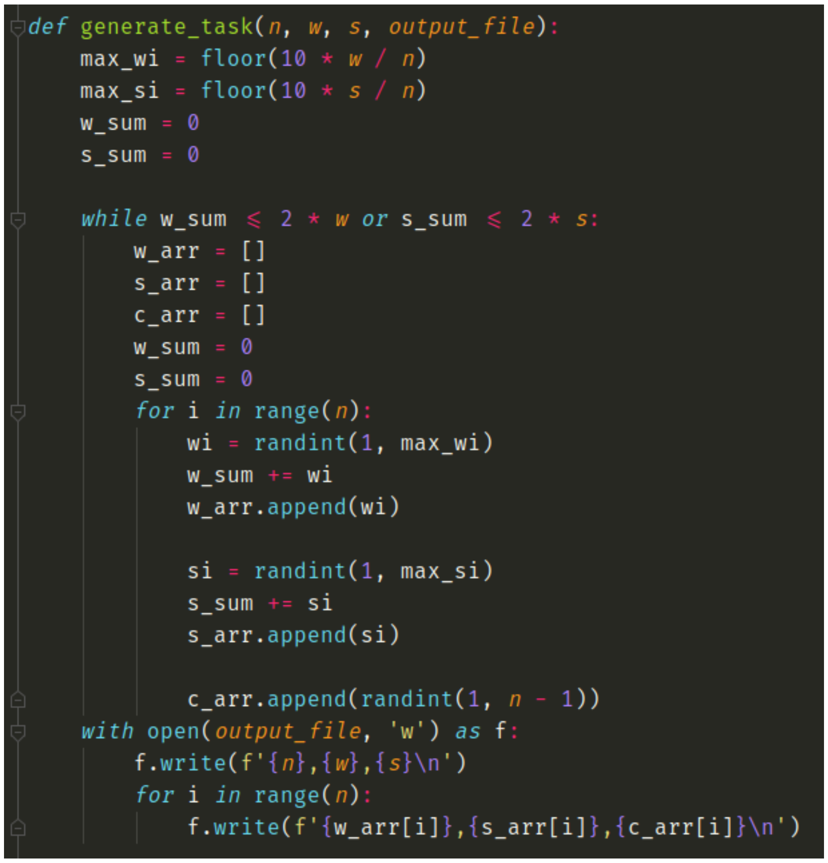
Introduction to Artificial Intelligence – Laboratory 1
Paweł Wiśniewski 232540

1 Source code

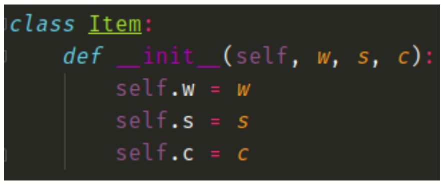
1.1 Task generator



```python
def generate_task(n, w, s, output_file):
    max_wi = floor(10 * w / n)
    max_si = floor(10 * s / n)
    w_sum = 0
    s_sum = 0

    while w_sum <= 2 * w or s_sum <= 2 * s:
        w_arr = []
        s_arr = []
        c_arr = []
        w_sum = 0
        s_sum = 0
        for i in range(n):
            wi = randint(1, max_wi)
            w_sum += wi
            w_arr.append(wi)

            si = randint(1, max_si)
            s_sum += si
            s_arr.append(si)

            c_arr.append(randint(1, n - 1))
    with open(output_file, 'w') as f:
        f.write(f'{n},{w},{s}\n')
        for i in range(n):
            f.write(f'{w_arr[i]},{s_arr[i]},{c_arr[i]}\n')
```

In the beginning, the function calculates the maximum weight and size for a single item. After that, it randomly chooses parameters for each item. The while loop repeats that process until the given criteria are met. My observations show that in almost all cases the requirements are satisfied after the first iteration. Finally, results are being written to the given file in the desired format.

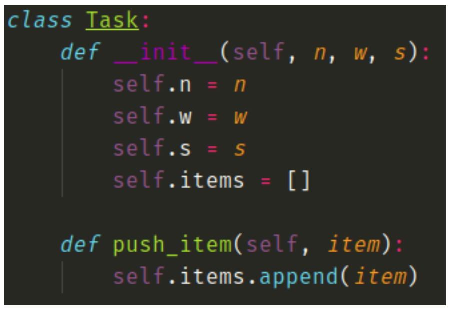
1.2 Item class



```
class Item:
    def __init__(self, w, s, c):
        self.w = w
        self.s = s
        self.c = c
```

Item class consists of a simple constructor, that saves informations about items weight, size and cost.

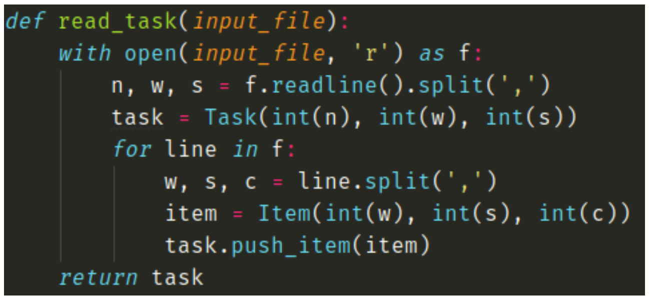
1.3 Task class



```
class Task:
    def __init__(self, n, w, s):
        self.n = n
        self.w = w
        self.s = s
        self.items = []

    def push_item(self, item):
        self.items.append(item)
```

Task class consists of a simple constructor, that saves informations about the number if items, maximum weight and size. In addition, it initializes an empty list designed for storing items. The class has a *push_item* method, that simplifies the insertion of items.

1.4 Task loading

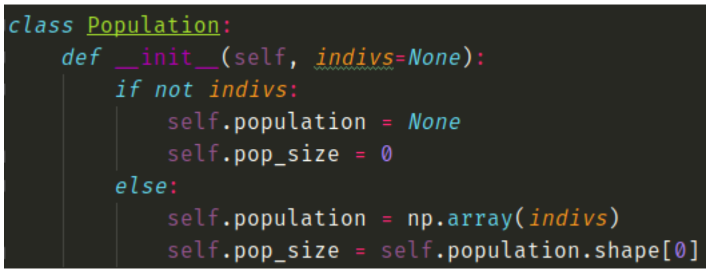


```
def read_task(input_file):
    with open(input_file, 'r') as f:
        n, w, s = f.readline().split(',')
        task = Task(int(n), int(w), int(s))
        for line in f:
            w, s, c = line.split(',')
            item = Item(int(w), int(s), int(c))
            task.push_item(item)
    return task
```

The task is loaded from a csv file. The first line contains informations about the number of items, the maximum weight and size. There are saved in an object of the Task class. The rest of the lines represents items and theirs parameters. The data are stored as the objects of the Item class and pushed to the task object. All the numbers are converted to integers.

1.5 Population class

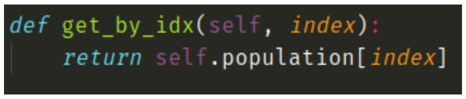
1.5.1 Constructor



```
class Population:
    def __init__(self, indivs=None):
        if not indivs:
            self.population = None
            self.pop_size = 0
        else:
            self.population = np.array(indivs)
            self.pop_size = self.population.shape[0]
```

The constructor uses a parameter *indivs* to pass a list of individuals. That list is converted into NumPy array (NumPy is a powerful package for scientific computing with Python). If no individuals are provided, an empty population is created.

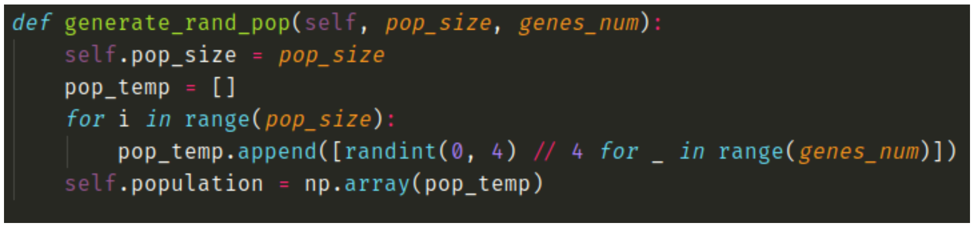
1.5.2 Get individual by id method



```
def get_by_idx(self, index):
    return self.population[index]
```

This method simplifies the process of obtaining the desired individual

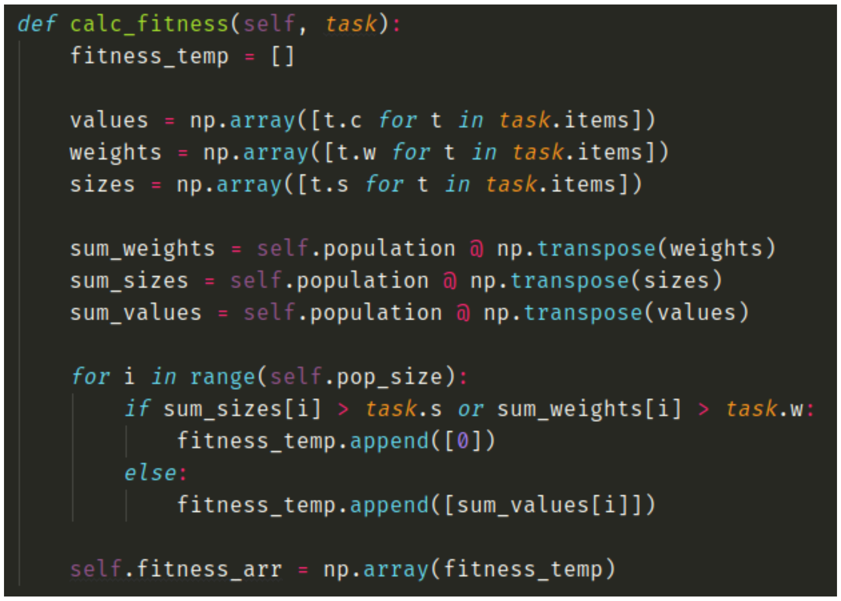
1.5.3 Generate random population method



```
def generate_rand_pop(self, pop_size, genes_num):
    self.pop_size = pop_size
    pop_temp = []
    for i in range(pop_size):
        pop_temp.append([randint(0, 4) // 4 for _ in range(genes_num)])
    self.population = np.array(pop_temp)
```

This method generates a *pop_size* number of individuals, each having *genes_num* genes. Genes informs which items are present in the knapsack. A gene can be either 0 (item absent) or 1 (item present). There is 80% chance to get 0 and 20% to get 1.

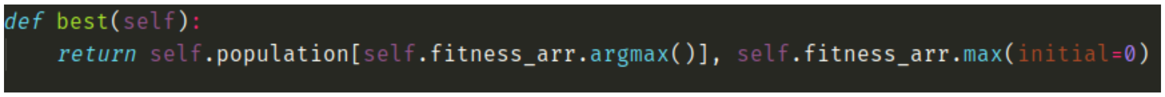
1.5.4 Fitness method



```python
def calc_fitness(self, task):
    fitness_temp = []

    values = np.array([t.c for t in task.items])
    weights = np.array([t.w for t in task.items])
    sizes = np.array([t.s for t in task.items])

    sum_weights = self.population @ np.transpose(weights)
    sum_sizes = self.population @ np.transpose(sizes)
    sum_values = self.population @ np.transpose(values)

    for i in range(self.pop_size):
        if sum_sizes[i] > task.s or sum_weights[i] > task.w:
            fitness_temp.append([0])
        else:
            fitness_temp.append([sum_values[i]])

    self.fitness_arr = np.array(fitness_temp)
```

At the beginning I split the items stored in the task object into three arrays representing values, weights and sizes. Then I do matrix multiplication, as a result of which I get three vectors informing about the total value, weight and size of items in the knapsack for each individual. Thanks to NumPy package, such calculations are carried out faster than other methods using loops. The fitness values for individuals meeting the task conditions are equal to the sum of the values of the items in their knapsacks. Individuals that exceed the maximum values have zeroed values. Results are assigned to the population objects, so that the tournament function will not be forced to repeat the calculations during each invocation.

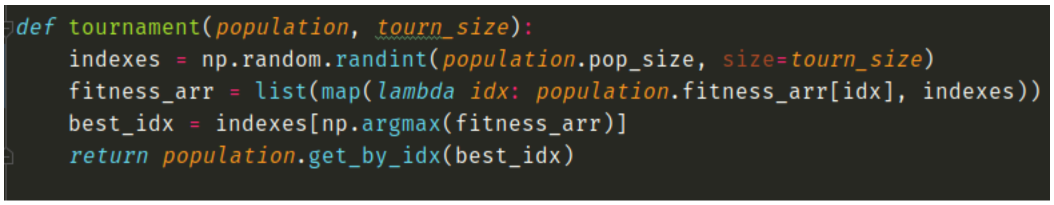
1.5.5 Get the best individual method



```python
def best(self):
    return self.population[self.fitness_arr.argmax()], self.fitness_arr.max(initial=0)
```

This method returns the best individual and his fitness value.

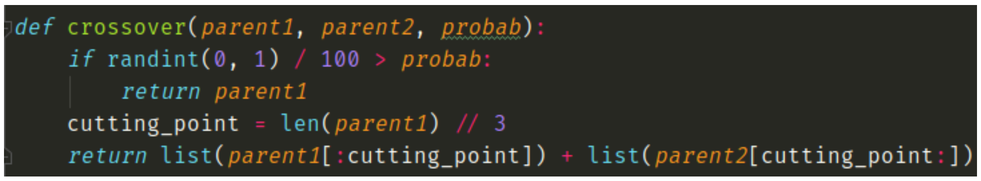
1.6 Tournament Function



```python
def tournament(population, tourn_size):
    indexes = np.random.randint(population.pop_size, size=tourn_size)
    fitness_arr = list(map(lambda idx: population.fitness_arr[idx], indexes))
    best_idx = indexes[np.argmax(fitness_arr)]
    return population.get_by_idx(best_idx)
```

This function selects a random slice from the population, retrieves fitness values for that subset and returns the best individual.

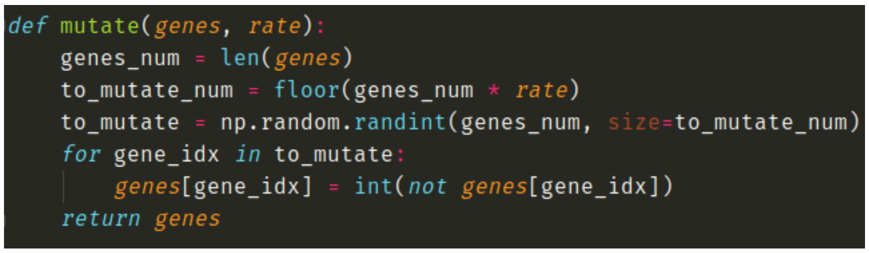
1.7 Crossover Function



```python
def crossover(parent1, parent2, probab):
    if randint(0, 1) / 100 > probab:
        return parent1
    cutting_point = len(parent1) // 3
    return list(parent1[:cutting_point]) + list(parent2[cutting_point:])
```

In the beginning a real number from the range [0, 1] is drawn. If that number is bigger than the given probability, no crossover occurs and the first parent is returned. Otherwise, a child is formed from clippings from both parents.

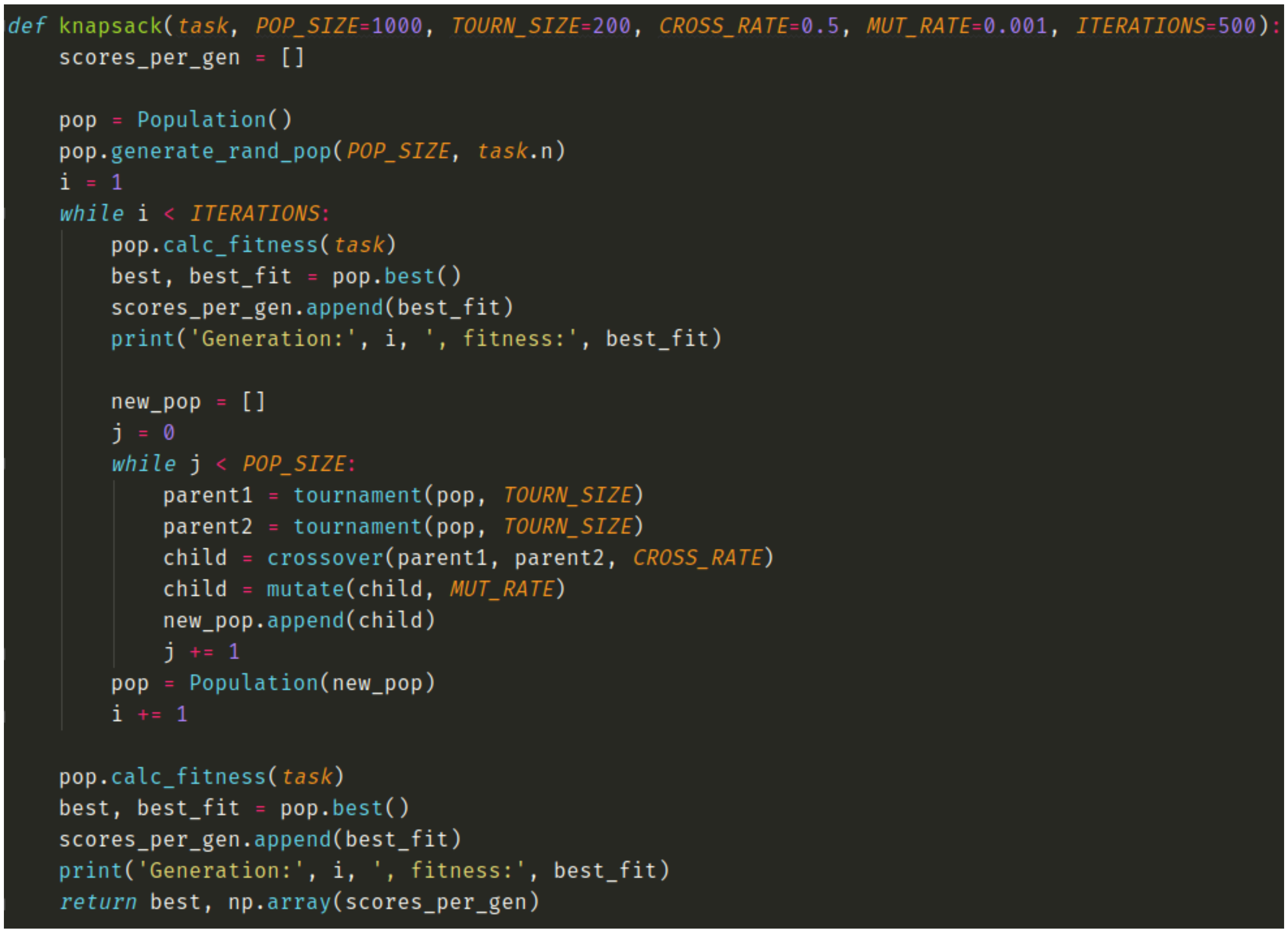
1.8 Mutation Function



```python
def mutate(genes, rate):
    genes_num = len(genes)
    to_mutate_num = floor(genes_num * rate)
    to_mutate = np.random.randint(genes_num, size=to_mutate_num)
    for gene_idx in to_mutate:
        genes[gene_idx] = int(not genes[gene_idx])
    return genes
```

The function selects random genes from an individual and converts their values to the opposite, then returns the mutated individual.

1.9 Mutation Function



```python
def knapsack(task, POP_SIZE=1000, TOURN_SIZE=200, CROSS_RATE=0.5, MUT_RATE=0.001, ITERATIONS=500):
    scores_per_gen = []

    pop = Population()
    pop.generate_rand_pop(POP_SIZE, task.n)
    i = 1
    while i < ITERATIONS:
        pop.calc_fitness(task)
        best, best_fit = pop.best()
        scores_per_gen.append(best_fit)
        print('Generation:', i, ', fitness:', best_fit)

        new_pop = []
        j = 0
        while j < POP_SIZE:
            parent1 = tournament(pop, TOURN_SIZE)
            parent2 = tournament(pop, TOURN_SIZE)
            child = crossover(parent1, parent2, CROSS_RATE)
            child = mutate(child, MUT_RATE)
            new_pop.append(child)
            j += 1
        pop = Population(new_pop)
        i += 1

    pop.calc_fitness(task)
    best, best_fit = pop.best()
    scores_per_gen.append(best_fit)
    print('Generation:', i, ', fitness:', best_fit)
    return best, np.array(scores_per_gen)
```

This function combines the others to find the best solution for the given knapsack problem. First, a random population is initiated. The creation of new generations will be carried out until the final condition is met. In this case the goal is to reach the given number of iterations. Two parents are selected using the tournament function. They are combined using the crossover function. A newly created child is mutated and stored in a new population. The best fitness value for each iteration is stored in the *scored_per_gen* list. The function returns the best individual from the last generation and the array *scored_per_gen*.

2 Analysis of the impact of the parameters