Instrukcja 3

Listy, pętle i kolizje w grze Snake

Podstawy tworzenia gier

Listy

Listy to języku Python dynamiczne tablice. W przeciwieństwie do tablic używanych w języku C++, listy nie wymagają wcześniejszej deklaracji rozmiaru (liczby elementów w niej przechowywanej), a także typu danych. Wewnątrz jednej listy możemy przechowywać elementy różnego typu (np. liczby całkowite, zmiennoprzecinkowe, napisy, itp., a nawet inne listy). Rozmiar listy dynamicznie się zmienia, co pozwala na bieżące dodawanie oraz usuwanie elementów.

1. W celu utworzenia listy należy wykorzystać nawiasy kwadratowe [ ]. Istnieje możliwość inicjalizacji pustej listy, lub zawierającej określone elementy.

   ```
   list1 = []                          #Pusta lista
   list2 = [1, "two", 3.14]            #Lista z 3 elementami
   ```

2. Listy posiadają wbudowaną metodę append służącą do dodawania nowych elementów do listy. Element zostanie dodany na końcu.

   ```
   list2.append("cztery")
   ```

3. Usuwanie elementów odbywa się za pomocą metody remove.

   ```
   list2.remove("two")
   ```

4. Aby wyświetlić wszystkie elementy listy wystarczy użyć jej nazwy w poleceniu print.

   ```
   print(list2)
   #Wyświetlone zostanie: [1, 3.14, "cztery"]
   ```

5. Python oferuje wbudowaną metodę `sort`, która umożliwia posortowanie listy. Dla napisów sortowanie jest wykonywane alfabetycznie, a dla liczb - w porządku rosnącym.

   ```
   numbers = [3, 1, 4, 5, 9, 2]
   numbers.sort()
   #Nowa kolejność listy to: [1, 2, 3, 4, 5, 9]
   ```

   UWAGA: Próba posortowania listy o mieszanych typach danych zakończy się błędem!

6. Listy obsługują „slicing" („wycinanie"). Pozwala on na dzielenie listy na mniejsze podzbiory. Należy przy tym zwrócić uwagę na numery elementów (liczone od zera) , a także podać przedziały do wycięcia. Rozpoczynając od początku listy możemy pominąć pierwszy numer elementu (lub ostatni w przypadku końca).

```
list3 = [1, 2.1, 3.7, "four", 5, "six"]
first_three = list3[:3]
#first_three = [1, 2.1, 3.7]
four_elements = list3[1:4]
#first_three = [2.1, 3.7, "four", 5]
```

UWAGA: Dolny zakres „przedziału” jest liczony WŁĄCZNIE, a górny NIEWŁĄCZNIE.

Pętla for ... in

Pętla for ... in jest używana w Pythonie do iteracji po elementach kolekcji (np. listy, krotki, słownika, ciągu znaków, itp.). Jest to jedna z najczęściej używanych pętli w języku Python. Ze względu na podobieństwo jest często porównywana do pętli for z języka C++, jednak jest od niej znacznie bardziej elastyczna i wygodniejsza w użyciu. Nie wymaga również zwiększania iteratora („licznika”) po każdym wykonaniu pętli, ponieważ robi to automatycznie.

Poniższy przykład przedstawia porównanie pętli w języku Python i C++

Python

```
for i in range(5):
    print(i)
```

C++

```
for (int i = 0; i < 5; i++) {
    cout << i << endl;
}
```

Obie pętle wyświetlają w konsoli liczby od 0 do 4, każda w osobnej linii:

```
0
1
2
3
4
```

Metoda range służy do tworzenia list składających się z określonej liczby elementów, w kolejności od zera. Zapis range(5) zastępuje ręczne utworzenie listy [0, 1, 2, 3, 4].

1. W celu iteracji po wszystkich elementach listy należy zadeklarować zmienną, która będzie reprezentować określony element w każdym wykonaniu pętli.

   ```
   examples = ["first", "second", "third"]
   for example in examples:
       print(example)
   ```

2. Nazwa zmiennej reprezentującej określony element listy w danej iteracji pętli jest dowolna. W poprzednim przykładzie wykorzystano nazwę example, jednak mogłaby to być dowolna inna nazwa na przykład x lub dla licznika tradycyjnie i.

Dobrą praktyką jest jednak stosowanie nazw nawiązujących do nazwy listy po której iterujemy. Dla listy items *(liczba mnoga)* nazwa pojedynczego elementu z listy, czyli item *(liczba pojedyncza)* będzie znacznie czytelniejsza od np. x.

```
for item in items
```

~~`for x in items`~~

3. Istnieje możliwość iteracji po określonych ciągach znaków. W tym celu zamiast listy podajemy napis.

```
for letter in "Python"
    print(letter)
```

Powyższy przykład wyświetli z osobna każdą z liter słowa „Python".

4. W przypadku gdy chcemy powtórzyć jakąś czynność określoną ilość razy, bez wykorzystania iteratora, możemy go pominąć wpisując w jego miejsce _ .

```
for _ in range(4)
    print("TEST")
```

Metoda range()

Metoda range służy do szybkiego tworzenia list składających się z szeregu określonego zakresu liczb.

1. Domyślnie range(n) tworzy listę składającą się z n elementów, zaczynając od zera. Na przykład range(5) zastępuje ręczne utworzenie listy

```
[0, 1, 2, 3, 4]
```

2. Podanie dwóch liczb do range(m, n) utworzy listę z zakresem liczb, której pierwszym elementem będzie liczba m (jest WŁĄCZNA w przedział). Ostatnim elementem będzie liczba n-1, ponieważ górna liczba zakresu jest NIEWŁĄCZNA. Na przykład range(2, 5) zastępuje ręczne utworzenie listy

```
[2, 3, 4]
```

3. Wykorzystując metodę range() każda kolejna liczba jest domyślnie większa o 1. Istnieje jednak możliwość zdefiniowania „kroku" o ile poprzednia wartość ma zostać inkrementowana. Służy do tego trzecia liczba, umieszczona ZA zakresem liczb. Użycie range(m, n, x) utworzy listę, której pierwszym elementem będzie liczba m, element ten będzie zwiększany o x, aż do osiągnięcia ostatniej możliwej liczby mniejszej od n. Na przykład range(10, 28, 5) utworzy listę:

```
[10, 15, 20, 25]
```

Snake

W poprzednim projekcie, położenie bloku (kwadratu) definiowano w następujący sposób:

```
block_size = 20
block_pos = [width // 2, height // 2]
```

gdzie block_size oznaczało długość boku kwadratu w pikselach, a block_pos tworzyło listę, której pierwszym elementem była współrzędna wysokości na ekranie, a drugim - współrzędna szerokości (kwadrat znajdował się na środku ekranu).

Powyższy kod można bardzo łatwo wykorzystać do przygotowania gry Snake. W tym przypadku, blok będzie reprezentował jeden „człon" węża. Jego rozmiar może pozostać bez zmian, ponieważ wąż będzie składał się z kilku „bloków".

Pozycja X i Y jednego bloku jest zapisana w formie listy. Ta lista może stać się częścią większej listy:

```
snake = [[100, 100], [90, 100], [80, 100]]
```

Lista snake składa się z 3 elementów - każdy z nich to położenie jednego z trzech bloków węża. W celu narysowania całego węża należy skorzystać z pętli for ... in , a w każdej iteracji pętli skorzystać z metody draw dla poszczególnego bloku.

```
for element in snake:
        pygame.draw.rect(game_view, "green", (element[0],
        element[1], block_size, block_size))
```

UWAGA: Każdy z członów węża (zapisanych w liście snake) w pętli for ... in stanowi jeden element. Pętla rysuje każdy z trzech elementów wykorzystując ich położenie X (element[0]) i Y (element[1]).

Wykorzystanie listy do zapisu położenia węża umożliwia jego prostą animację. W poprzednim projekcie, w trakcie każdej iteracji pętli gry obliczano nowe położenie bloku i zapisywano w formie listy o nazwie block_pos. W przypadku gry Snake, nowe położenie może zostać potraktowane jako „głowa" węża, która zostanie dodana na początek listy snake. W ten sposób, zamiast przesuwać cały obiekt, blok staje się najpierw głową węża [0], następnie (po dodaniu nowej głowy) jego pierwszym członem [1], drugim członem [2], itd. Poniższy kod dodaje listę new_head na początek (położenie 0) listy snake.

```
snake.insert(0, new_head)
```

Obecny stan projektu ma podstawową wadę - wąż jest wydłużany w nieskończoność, ponieważ w każdej iteracji pętli zostaje dodany nowy element. Konieczne jest zatem każdorazowe usunięcie ostatniego elementu listy snake (głowa jest dodawana na początek), aby wąż zawsze liczył taką samą ilość elementów. Najłatwiej jest to zrobić metodą pop(), która służy do usuwania ostatniego elementu listy.

```
snake.pop()
```

Kolizje

Aby gra w Snake'a była funkcjonalna, konieczne jest dodanie elementu rozgrywki w postaci zbierania punktów i wydłużania węża. Jedzenie (food) o losowym położeniu w oknie gry i o rozmiarze jednego bloku sprawdzi się do tego idealnie.

```
food = [random.randint(0, (WIDTH // block_size - 1)) *
       block_size, random.randint(0, (HEIGHT // block_size - 1))
       * block_size]
```

Powyższy kod tworzy listę o nazwie food i (na takiej samej zasadzie jak w blokach węża) umieszcza w niej losowe położenie na szerokości i wysokości ekranu. Warto jednak rozwinąć ten kod. Istnieje ryzyko, że wylosowane położenie pokryje się z położeniem węża - w takiej sytuacji należy powtórzyć losowanie. Poniższa funkcja przeprowadza losowanie prawidłowej lokalizacji dla jedzenia i zwraca je jako listę.

```
def generate_food(snake):
    while True:
        food = [random.randint(0, (WIDTH // block_size - 1)) *
              block_size, random.randint(0, (HEIGHT // block_size
              - 1)) * block_size]
        if food not in snake:
            return food
```

UWAGA: Instrukcje warunkowe if w języku Python umożliwiają łatwe sprawdzenie, czy określony element (food) znajduje się lub nie znajduje się w liście (snake). Zamiast operatorów stosuje się w tym przypadku zwroty in lub not in.

Funkcja generate_food powinna zostać wywołana na początku gry - położenie jedzenia jest stałe, nie zmienia się w trakcie każdej iteracji pętli. Kolizja głowy węża z jedzeniem oznacza takie samo położenie dwóch bloków, a dokładniej - identyczna zawartość listy food i snake[0] (lista głowy węża). Można to wykorzystać w części snake.pop(), ponieważ kolizja powoduje wydłużenie węża i ponowne wylosowanie położenia bloku jedzenia - w takiej sytuacji nie ma konieczności usuwania ostatniego elementu listy snake, a lista stanie się dłuższa o 1 element.

```
if snake[0] == food:
    food = generate_food(snake)
else:
    snake.pop()
```

Zadanie

Wykorzystaj projekt z poprzedniego zadania. Napisz grę Snake, bazującą na fragmentach kodu z instrukcji. Zmodyfikuj proces przesuwania bloku w ten sposób, aby możliwe było uzyskanie nowego położenia głowy węża. Zaimplementuj kolizje węża z blokiem jedzenia. Obsłuż próbę przesunięcia węża poza okno gry (granice mapy) - na przykład poprzez instrukcje warunkowe dotyczące położenia głowy.

Wskazówka: Pamiętaj o „narysowaniu" bloku jedzenia!

ROZSZERZENIE: Warunkiem zaliczenia części rozszerzonej jest wykorzystanie poprzedniego zadania w wersji rozszerzonej (ciągłe przesuwanie węża). Wykonaj część podstawową. Zaimplementuj kolizje węża z samym sobą - w takiej sytuacji gra ma zostać zakończona, a w konsoli ma pojawić się informacja o zebranej liczbie punktów. Wykorzystaj kod z poprzedniego zadania, aby kolor bloku jedzenia był losowany przy wywołaniu funkcji generate_food.

Opcjonalnie: Dodaj więcej jedzenia.

Opcjonalnie[2]: Nie ma możliwości, aby wylosowano położenie istniejącego już na mapie jedzenia (lista + pętla).

Opcjonalnie[3]: Przeszkody, które w przypadku kolizji skracają węża (a może kończą grę, odejmują punkty, itp. - wybór zależy od Ciebie, bądź kreatywny!).